MPW 3.2 C
Release Notes

The primary changes that have been made to the C compiler for MPW 3.2 are for the purpose of optimizing code size and performance. Other important changes are:

- An option to control the level of optimization.
- An option to control the level of warning messages.
- An option for machines having the MC68020 and up that allows stand-alone code segments greater than 32K.
- An option for invoking the "32-bit everything" run-time architecture.
- Provision for passing parameters to inline functions via registers by using `#pragma parameter`.
- Support for the MacApp debugger,and code profilers, controlled by a command line option and a `#pragma`.
- A pragma to prevent in an efficient manner the multiple inclusion of header files.
- A pragma to force the generation of MC68020 code.
- Prevention of dead code stripping when linking by use of `#pragma force_active`.
- Several other pragmas comparable to or related to existing command-line options.

-opt

This option controls the application of optimizations to the generated code. The syntax is:

```
-opt param-list
```

where

```
param-list = param[,param]

param = exclusive | nonexclusive

exclusive = on | off | full          # choose one at most
```


Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

```
nonexclusive = nopeep | nocse        # choose zero or more
```

The meanings are as follows:

`off`	no optimizations
`on`	"normal" optimizations
`full`	include register allocation
`nopeep`	inhibit peephole optimization
`nocse`	inhibit the elimination of common subexpressions

The default is `-opt on`. The choice `-opt full` causes any free registers in the temporary storage group (A0-A2, D0-D3) to be used for the storage of variables. This choice increases compile time and may or may not significantly reduce execution time.

-warnings

The following command-line options replace the former options `-w` and `-w2`:

`-warnings on`	normal warning level, the default
`-warnings full`	maximum warning level
`-warning off`	no warnings

Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

-bigseg

The `-bigseg` command-line option allows code segments to grow beyond the 32K limit. It uses the 68020 PC-relative modes of addressing with 32 bit offsets. This obviously limits code produced with this option to machines with a 68020 or higher numbered CPU. The advantage is that it can produce code which is entirely contained within a single large segment and which requires no runtime overhead of address fixup (possibly not even a jump table). This can be especially useful in porting UNIX tools written for systems which know no artificial limits on code size or for developing large Macintosh CDEVs or desk accessories. The only other requirement for these code resources is that all externally defined entries must be within the first 32K of code.

-model

The following options have been added to support "32-bit everything," the run-time architecture that removes the 32K restriction on segment size, jump table size, and the size of the global data area:

`-model near`	the default
`-model far`	remove the 32K restriction for both code and data
`-model nearData`	the default
`-model farData`	remove the 32K restriction for data
`-model nearCode`	the default
`-model farCode`	remove the 32K restriction for code

For further details, see MPW 3.2 Run-Time Architecture Release Notes.

#pragma parameter

This is used to specify that parameters for inline functions be passed directly in registers. The syntax is shown below:

```
Reg ::= __D0 | __D1 | __D2 | __A0 | __A1
```


Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

```
#pragma parameter [Reg] functionname [(Reg[,Reg])]
```

It is important to note that the only registers that may be specified are `D0`, `D1`, `D2`, `A0`, and `A1`. These registers are considered to be temporary registers used internally by the C compiler. Thus, if any function is using those registers and then makes another function call, the calling function will generate save and restores of the affected registers around the call. This knowledge allows the safe use of those registers within the inline function.

The `#pragma parameter` directive must precede the inline function declaration, designated by `functionname`, to which it applies. The size and order of parameters is determined by the function prototype alone. The pragma then directs those parameters to be placed within the specified registers. The first register, which optionally may be omitted, determines where the function result will be returned. Next appears the required function name. It must agree with the name of the inline function definition to which it applies. This is optionally followed by a list of registers enclosed within parenthesis. This list contains one or more registers separated by commas. The leftmost function parameter will be assigned to the first register listed in the parenthesis. The remaining parameters will be assigned in order to the remaining listed registers. The number of registers requested need not match the number of parameters in the function prototype; extra listed registers will be ignored. When there are more parameters than specified registers, the extra parameters will be placed on the stack as they normally would have been in the absence of the `#pragma parameter` directive.

Other than the actual placement in registers, these parameters will be assigned in accordance with the either the normal C or Pascal parameter-passing conventions. That is: Pascal style functions will assign parameters of the size specified, from left to right, first to registers and then pushing the remainder on the stack. C style parameters will always be promoted to size int and then placed on the stack from right to left until the remaining parameters match the number of specified registers. Those leftmost parameters will then be placed, also as ints, in the correspondingly ordered set of registers.

The function return will also be handled by normal conventions: Functions returning type void will have no return regardless of whether or not an initial register was specified. For other return types with a specified return register, the Pascal statements will move correct-sized results from the register, while those returned from standard C functions will be of size int. If there is no initial register specified, the default rules will apply: Pascal style routines will return appropriately sized function results from the stack, while C style routines will return int sized values in D0.

This pragma can be used in the C interfaces to allow some Macintosh calls to be rewritten as inline functions requiring no glue. An example would be:

```
#pragma parameter __A0 NewHandle (__D0)
pascal Handle NewHandle (Size logicalSize) = 0xA122;
```


Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

The compiler will generate code to move logicalSize directly into D0. The entire body of the function will be the inline trap $A122. Finally, the result will be moved directly from A0 to the assignment variable. Rewriting these interfaces will result in smaller, faster code when compared to the glue which it would be eliminating.

-trace and #pragma trace

`-trace` and `#pragma trace` will turn on support for the MacApp debugger or for a code profiler to give performance information. When this feature is on, the C compiler will insert a function preamble and postamble into the code. The preamble will be: `JSR %_BP` inserted after the `LINK` instruction on entry into the function. The postamble is: `JSR %_EP` inserted just before the `UNLK` instruction on function exit. The `%_BP` and `%_EP` calls, which take no parameters, must then be provided by the programmer to support either MacApp debugging or profiler support.

The "`#pragma trace on|off`" option is placed in source code to turn this feature on or off for an individual function only. The preferred location for the `#pragma` would be immediately before the function definition. However, it can be recognized while in the body of the function itself.

- Note that all pragmas relating to a function are in effect collected and processed before the code of the function is processed, so that only that last declaration of `trace on` or `trace off` for a given function will be honored.

The command line option "`-trace on|off|always|never`" provides more control. The `on` and `off` options provide the default setting, which can then be overridden by `#pragma trace` statements embedded within the source code. Choosing `always` or `never` overrides the #pragma settings, respectively forcing all functions to either have the function pre- and post-ambles or not.

#pragma once

The #include "once" feature provides an efficient way of inhibiting multiple includes of the same file. We currently handle this by inserting a sequence of the form:

Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

```
#ifndef name
      #define name
      ...
      #endif
```

inside of an include file to suppress inclusion of the main body of the file more than once. This, however, requires the compiler to open the file *each* time (applying the appropriate search rules), and to scan through the entire include file looking for the delimiting `#endif`. This time consuming process can be considerably reduced by providing a mechanism whereby the compiler maintains a list of the files that have been #included and takes the responsibility of inhibiting multiple inclusion.

The "once" feature is assumed by the compiler recognizing either of the following conditions in a #include file:

- The presence of `#pragma once`. This unconditionally indicates the file is to be included only once.
- The presence of an initial `#ifdef`, `#ifndef`, or `#if`, and the delimiting `#endif` surrounding an include file (ignoring comments). Since there may be times when a user would *not* want to have the compiler automatically suppress multiple inclusion, the option `-notonce` has been provided. However, if the user has explicitly inserted `#pragma once` in a file, the `-notonce` option is overridden for that file.

The advantage of the pattern recognizer is that the use of conditionals surrounding the contents of include files to suppress reprocessing is a common practice. Indeed, it is what is presently done in all our CIncludes with the exception of Assert.h. Thus, no changes to our CIncludes need to be made in order to use the "once" enhancement.

The #pragma complements the pattern recognizer. It can always be used. Also, it is compatible with some other existing C compilers (e.g., GNU GCC). The word "once" was chosen to attain this compatibility.

#pragma processor

The choice between generating 68000 code and 68020 code can be made by use of the pragma

```
#pragma processor 68000|68020.
```

The default, of course, is to generate 68000 code.

Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

#pragma force_active

The syntax of this pragma is

```
#pragma force_active on|off
```

The default is "off." The effect of the pragma with the parameter "on" is to prevent the Linker from deleting the code within the scope of the pragma in the event that there are no references to it, an action known as "dead code stripping."

#pragma push, #pragma pop

This pair of pragmas saves (`#pragma push`) and restores (`#pragma pop`) the conditions set by the options (or pragmas) `-s`, `force_active`, `-mbg`, `-warnings`, `processor` (or `-mc68020`), and `-opt`.

mbg, warnings, and opt

The following three pragmas correspond exactly to the options of the same names:

```
#pragma mbg off|full|on|ch8|0...255
      mbg ch8         # v2.0 compatible macsbug symbols
      mbg off         # no macsbug symbols in the code
      mbg on|full     # full macsbug symbols
      mbg <n>         # macsbug symbols to length <n> (<n> can be 0..255)

#pragma warnings on|off|full  # set warning level; "on" is the default

#pragma opt off|on|full
      opt off         # don't apply code optimizations
      opt on|full     # choose level of code optimization (on is default);
                      #  can modify with [,nopeep] [,nocse]
                      # (no peephole, no common subexpression)
```


Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

Linking Requirement for 881

△ **Important** The library CLib881.o has a new positioning requirement as a consequence of the elimination of CRuntime.o. CLib881.o must now not only precede all members of {CLibraries} in the link sequence but must also precede the library Runtime.o. △

Changes to C Library and Interfaces

Libraries

ANSI C

The MPW 3.2 C libraries are largely in conformance with the current ANSI C standard (X3.159—1989).

File Types

If you create a new file and do not specify that it is a binary file by including `'b'` as part of the open mode of an `fopen()` call or `F_BINARY` as part of the mode for an `open()` call, the file that is created will be of type TEXT. Except for marking a file as a TEXT file at its creation, there is no difference in the way that MPW C handles text and binary files internally.

There is also a new function which can be used to set the creator and file type of any file. The function is:

```
void fsetfileinfo (char *filename, OSType newcreator,
                                   OSType newtype);
```


Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

cfree()

Because `cfree()` is not part of the ANSI standard this function has not been supported since the release of MPW 3.0. The code for `cfree()` has now been removed from the libraries. If you have any existing code which uses this function, you can create your own macro or function to replace it. Since MPW C has always ignored the second two arguments of `cfree()`, the function call `free(p)` is equivalent to `cfree(p, int1, int2)`.

dup()

Because `dup()` is not part of the ANSI standard and because its functionality is duplicated in the `fcntl()` function, the `dup()` function is no longer supported and will be removed from a future version of `StdClib.o` in the MPW C Libraries. The function call `fcntl(filedes, F_DUPFD, 0)` is exactly equivalent to the call `dup(filedes)`. Existing code should either be changed to call `fcntl()` directly or a macro or local definition of `dup()` should be provided by the user.

fputs()

This function will now set *errno* when applied to a read-only file.

malloc()

`malloc` is currently limited to a maximum size of about 8 MB per call. Size parameters of value 2^{23} or greater result in a null pointer being returned. This is not a new limitation, but it was not previously documented.

Interfaces

String Pointer Parameters

The definition `typedef const unsigned char *ConstStr255Param` has been introduced in types.h for compilers that support `const`. It is intended to replace the use of `const Str255`. For consistency, similar declarations have been included for `ConstStr63Param`, `ConstStr32Param`, `ConstStr31Param`, `ConstStr27Param`, and `ConstStr15Param`.

Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

Generic Pointers

In function declarations, all instances of `Ptr` as an argument type have been changed to `void *`. This removes the restriction that parameters represented by pointers had to be of type `char *`. It is intended in the near future to make the equivalent change for Pascal by using `UNIV Ptr`.

Fcntl.h

Definitions for the following "internal-use only" macros have been removed from the header file FCntl.h.

```
O_TMP
O_USEP
```

IoCtl.h

The definition of the internal struct type `_SeekType` has been removed from IOCtl.h.

Time.h

`#define CLK_TCK 60` has been changed to `#define CLOCKS_PER_SEC 60`.

The type cast in the `#define` for `difftime` has been changed to `long double`.

Types.h

The type `Str32` has been moved from AppleTalk.h to Types.h.

Known Outstanding Bugs

Extra link file

When pointers are subtracted, we do a divide of the difference by the size of structs being pointed to in order to return the number of elements between the pointers. Since we are now forcing all of our arithmetic to go through int sized operators, this divide becomes a long divide. No longer fitting the DIV.W instruction in the 68000, we call a library routine: LDIVT. You now will be required to link against CRuntime.o, even if your code did not previously need to. Sorry, this is another one we'll correct.

Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

Initializer Type Checking

Type compatibility is not checked to the same extent for initializers as it is for ordinary assignments. If there is any doubt about the correctness of an initializer, it is safer to write the declaration without an initializer and to use a subsequent assignment statement to perform the initialization.

Incomplete Enum Declaration

Although it is legal C syntax to declare an enum with no constant values, MPW 3.2 C treats such a declaration as an incomplete type specifier and returns an error message. Be sure to declare *enums* fully. This is reported as a bug inasmuch as it is a deviation from the ANSI Standard.

Assignment from Floating Point Arrays to Integer Types

When a floating point number from an array is assigned to an integer type, a library routine converts the floating point number to an integer and returns it in register D0. Unfortunately, a previously calculated array offset is placed in D0 before the actual assignment can be made.

The work-around is to assign the variable from the array to an intermediate non-array variable of the same type, and then to assign the intermediate variable to the one of integer type.

Example:

```
extended e, f[3];
 int i, j;

 i = f[j];  /* generates incorrect code */

 e = f[j];  /* generates correct code */
 i = e;
```

Assignment from Fields of Structs

If an assignment is made of a field of a struct being returned, the compiler cannot properly apply the field's offset and may assign the wrong portion of the struct.

The work-around is to assign the returned struct to an intermediate variable of the same type. Then assign the proper field from that intermediate copy to the desired variable.

Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

Example:

```
Point GetPoint(void);

main()
{
        Point pt;
        short y;

        y = GetPoint().v; /* generates incorrect code */

        pt = GetPoint();  /* generates correct code */
        y = pt.v;
}
```

Pascal-Style Functions Returning Structs

When a Pascal-style function returning a struct is used in a complex expression, the offset to the struct may be miscalculated.

The work-around is to return the struct directly to an intermediate variable, and then to use that variable in the desired expression.

Example:

```
pascal Rect PGetRect(void);

main()
{
        Rect r, *pr;

        pr = (r = PGetRect(), &r);     /* generates incorrect code */

        r = PGetRect();                /* generates correct code */
        pr = &r;
}
```

Floating Point Parameters in Pascal-Style Functions

Pascal-Style C functions may corrupt the value of floating point parameters. The best advice at present is to use only type *extended* as parameters in Pascal-style functions. This is because the bug appears to be in the process of conversion to extended, which is the type used for floating point arithmetic.

Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

Example:

```
pascal void FTest(float kp)
{
      printf("%f\n", kp);      /* usually prints wrong value */
}
```

-model

The code to distinguish the cases `-model farData` and `-model farCode` is not yet in operation. The use of either of the above options has the same effect as using `-model far`.

#pragma parameter

The use of complex expressions as parameters may cause a fatal compiler error with the message: "Expression too complicated."

Large data

When compiling with the -m option (to address data greater than 32k), the code generator emits code that calculates a variable's address by copying A5 into another A register and then adding the appropriate offset. In expressions involving multiple array indices, the code generator may not have a spare A register to use for the address calculation. In these cases the code generator will abort with a message like:

```
### Code Generator Fatal Error
### Register 8
### Within f (Error 2001): Expression too complicated, code
                           generator out of registers
```

We have reduced the frequency of these aborts, but they do still occur. If this happens to you, try to identify and simplify the offending expression.

Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

Pascal Keyword

The pascal keyword is broken in the specific situation where one attempts to call a C function which returns a pointer to a pascal-style function. The C compiler misinterprets the C function as a pascal-style function and the function result is lost.

Declaration of void types

The compiler incorrectly allows variables of type `void` to be declared. (Variables of type `void *` are ok.)

Incomplete arrays

The compiler allows incomplete arrays to be declared as variables and within struct definitions. Not only is this incorrect, but relying upon it and then indexing off the result can yield unpredictable results.

Floating Point

Floating point arithmetic is handled separately from integer arithmetic and still suffers from some of the same code generation problems that have already been fixed in the integer cases. Pascal functions of type extended or float do not return the correct value.

Unsigned Constants

Any unsigned int or unsigned long constant above `0xFFFF7FFF` (which is declared either by a trailing `u` on the number or an explicit cast to `unsigned`) is treated as a signed constant when used in an arithmetic expression. This means, for instance, that dividing by such a constant will be done using a signed division operation instead of an unsigned division.

Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

The following code illustrates the problem and shows the workaround:

```
#define UINT_MAX 0xFFFFFFFFu
main()
{
      int p, x;
      x = 2;
      p = UINT_MAX / x; /* Wrong result: p = 0 */
      p = UINT_MAX / (unsigned) x; /* Right result: p = 0x7FFFFFFF */
}
```

The Macro Preprocessor

The macro preprocessor is not a complete ANSI implementation in the sense that the rules for rescanning and replacement in the context of ## are not observed. For example:

```
#define glue(a,b) a ## b
#define HIGHLOW "hello"
#define LOW LOW", world"
glue(HIGH, LOW)
```

results in: "hello"", world"; rather than the correct: "hello".

String concatenation doesn't work when the second string is the result of expanding a macro. For example:

```
#define REASON "I felt like it"
char *message = "Program aborted because " REASON;
```

results in an error at the token REASON.

Circular macro definitions will hang or crash the compiler.

`#include <FileName>` works incorrectly. If `FileName` isn't found in the standard places (`{CIncludes}` and directories specified as searchpaths on the command line), the compiler will search the current directory for it.

`#include ":dir:FileName"` looks for `:dir` relative to the current directory, and not relative to the directory of the file containing the `#include`.

Many reports of macro expansion errors, other than the ones just mentioned, have so far turned out to have been code generation errors produced by the expanded source code. Macros are good at expanding to more complex code which will require temporary values to be stored on the stack; this is just the area of code generation that we are most likely to have problems with.

Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

C Language Library and Interface

In `printf()`, the "`0`" character flag works incorrectly for the "f" (floating point) conversion specifier. It will NOT pad with leading zeros.

Bug Fixes

The items listed below are bugs that existed in MPW 3.0 or MPW 3.1. The list does not include bugs that first appeared in Alpha or Beta versions of MPW 3.2.

- The ANSI error directive, #error, used to work in contexts like

  ```
  #error "Ugh, this code is terrible"
  ```

 but not if you merely said

  ```
  #error
  ```

 Both forms now work. In the future they will trigger a fatal error condition which stops compilation immediately; for now, they will report the error and allow the compiler to hunt for more errors.

- The following two fixes were to facilitate compilations from C++ source: the maximum number of nesting levels was increased to 48 and the maximum length of identifiers was increased to 250 characters.

- The keyword `entry` was removed to conform with the ANSI specification.

- Conditionals such as (anyVar < 1) or (i == y) are now handled properly when assigned directly to fields within a structure. The specific bug reports we had were for assigning to a field that is either a bitfield or an indexed structure e.g.:

  ```
  s[i].f2 = (anyVar < 1);
  ```

Other Useful Information

The following properties of MPW C have caused problems for users who were unaware of them. As far as we know, they are consistent with the ANSI Standard.

Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

"Old C" Style Function Declarations

Parameter checking (at function call) is done only for function definitions that use the prototype form:

```
int foo(int parm1, int parm2)
```

and not for definitions using the "old C" style:

```
int foo(parm1, parm2)
int parm1;
int parm2;
```

In addition, if prototype declarations and "old style" definitions are mixed, no parameter checking will be done beyond the "old style" definition.

Pascal Keyword

The "pascal" keyword for function declarations in MPW C does not do everything that the Pascal compiler does for its function declarations. The pascal keyword in function declarations results in the name being passed to the linker in all upper case letters, parameters are reversed on the stack from the normal C manner, and they are placed there in the size actually used rather than int. The Pascal compiler additionally passes parameters that are records greater than 4 bytes long as pointers to the record. For C parameters (even in functions declared to be pascal style), entire structs can be passed on the stack. Therefore, to communicate with real Pascal functions, such a parameter must be explicitly declared as a pointer to the struct. Return values that are records greater than 4 bytes are a harder problem to handle. Not only will the Pascal compiler return a pointer to the record, but it will automatically allocate room for the actual returned record. No matter if we declare the C function to return a struct or a pointer to the struct, the calling routine will not automatically provide storage for it. Thus, such pascal type functions cannot be created in C without providing some type of assembly language glue. This was a design decision made in the early days of MPW C and is currently under review.

Load/Dump

Only one `#pragma load` directive is allowed per compiled unit.

Note that `#pragma dump` saves only symbol information. Since initializers actually generate code, we cannot load statements such as:

```
static const Boolean done = false;
```


Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

Character Comparison

The user should be aware that character literals, e.g. `'∂'` and elements of string literals, e.g. `a[1]`, where char a[] = `"a∂b"`, are signed quantities, and will be sign extended before a (long) compare to a char variable is performed.

Contradictory statements to be found in earlier MPW C Release Notes are erroneous and should be ignored.

//-Style Comments and Backslash-Carriage Return

Macro definitions can continue over multiple lines by escaping the carriage return with a backslash character. //-style comments run from the // characters to the end of the line. So what should the behavior be when a backslash-CR combination appears at the end of a line containing a // comment? This turns out to be an incredibly controversial issue, with strong arguments on both the "Continue the comment" and "Terminate the comment" sides. So be warned: the behavior is, at least for now, officially undefined and subject to change in future releases.

Unterminated Strings as Macro Text

Macros of the form

```
#define UNCLOSED_STRING "There's no closing quote
```

or

```
#define UNCLOSED_CHAR 'x
```

are automatically closed at the end of the line; the single or double quote, as is appropriate, is implicitly added.

Copyright Apple Computer, Inc.
1990-1991. All rights reserved.

Macintosh®

Macintosh Programmer's Workshop 3.0 C Reference

APPLE COMPUTER, INC.

This manual and the software described in it are copyrighted, with all rights reserved. Under the copyright laws, this manual or the software may not be copied, in whole or in part, without written consent of Apple, except in the normal use of the software or to make a backup copy of the software. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to another person. Under the law, copying includes translating into another language or format.

You may use the software on any computer owned by you, but extra copies cannot be made for this purpose.

The Apple logo is a registered trademark of Apple Computer, Inc. Use of the "keyboard" logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

© 1988 Apple Computer, Inc.,
20525 Mariani Avenue
Cupertino, CA 95014
(408) 996-1010

Apple, the Apple logo, LaserWriter, Macintosh, APDA, and SANE are registered trademarks of Apple Computer, Inc.

© AT&T, Inc., 1988

DEC, VAX, and PDP are trademarks of Digital Equipment Corporation.

IBM is a registered trademark of International Business Machines Company.

Intel is a registered trademark of Intel Corporation.

ITC Garamond and ITC Zapf Dingbats are registered trademarks of International Typeface Corporation.

Microsoft is a registered trade-mark of Microsoft Corporation.

POSTSCRIPT is a trademark of Adobe Systems Incorporated.

Linotronic is a registered trademark of Linotype company.

Adobe Illustrator 88 is a trademark of Adobe Systems Incorporated.

ImageStudio is a trademark of Esselte Pendaflex Corporation in the United States, of LetraSet Canada Limited in Canada, and of Esselte LetraSet Limited elsewhere.

QMS is a registered trademark of QMS, Inc.

Motorola is a trademark of Motorola, Inc.

NS16000 is a trademark of National Semiconductor Corporation.

UNIX is a registered trademark of AT&T Company.

Z8000 is a trademark of Zilog Corporation.

Simultaneously published in the United States and Canada.

MPW sample programs

Apple Computer, Inc. grants users of the *Macintosh Programmer's Workshop* a royalty-free license to incorporate *Macintosh Programmer's Workshop sample programs* into their own programs, or to modify the sample programs for use in their own programs, provided such use is exclusively on Apple computers. For any modified *Macintosh Programmer's Workshop* sample program, you may add your own copyright notice alongside the Apple copyright notice.

Contents

Figures and tables xi

Preface xiii

What this manual contains xv
Other reference material you'll need xvi
Typographic and spelling conventions xvii
Language notation xvii
Boldface xviii
Spelling and capitalization xviii
Aids to understanding xix
For more information xix

1 About MPW C 1

About the C compiler 3
About MPW C version 3.0 3
Floating-point enhancements 4
MC68020 enhancements 4
The Macintosh interfaces 4
About the C libraries 5
About the C examples 7
Installing MPW C 9
Using header files 9
Segmentation control 11
Creating resources 13
Creating an application in C 13
Building an Application 13
Compiling an application 14
Linking an application 15
Creating an MPW tool in C 17
Building a tool 18
Compiling a tool 18

Linking a tool 19
Creating a desk accessory in C 19
Desk accessory restrictions 19
The DRVRRuntime library 20
Desk accessory routines 21
Building a desk accessory 21
Compiling a desk accessory 22
Linking a desk accessory 22
Stand-alone code resources 23
Creating code for different models of Macintosh 24
Source code 24
Header files 24
Compiler options 25
Creating code for the MC68881 and MC68020 25
MC68881 enhancements 25
Linking with the `-mc68881` option 26
The `extended` data format 27
MC68881 instructions 27
Floating-point registers 27
MC68020 enhancements 27

2 The MPW C Language 29

Language definition 33
Lexical conventions 34
Keywords 34
Character constants: newline, carriage return, and vertical tab 34
Ellipsis character 35
Identifiers 35
Numeric constants 35
String constants 36
Pascal-style strings 36
String concatenation 37
Line-end comments (Apple extension) 37
The preprocessor 37
The `#include` directive 38
Reserved symbols 38
Conditional compilation 39
Predefined symbols 39
Pragmas 40

Segmentation 40
Dumping and loading precompiled information 41
Unused variables and parameters 41
Stringization 42
Tokenization 42
Types 43
Scalar data types 43
Integral types 44
Pointer types 45
SANE types 45
Type `void` 45
Enumerated types 46
Structures 46
Bit-fields 47
Type qualifiers 47
`const` 47
`volatile` 48
`pascal` 50
The `register` class specifier 50
Integer conversions and arithmetic operations 50
Sequence points 51
Evaluation order 52
Function declarations and definitions 53
Pascal-style functions 55
Calling Pascal-style functions 55
Writing Pascal-style functions 56
Declaring and using Pascal-style function pointers 57
Direct functions 57
Parameters to the `main` function 59
Compiler implementation details 59
Size and byte alignment of variables 60
Byte ordering 60
Variable allocation and register use 61
Bit-fields 61
Bitwise operations 62
Compiler limitations 62
Structure results 63
Multiple external definitions 63
Standard Apple Numerics Environment extensions 63
Expressions 66

Comparisons involving a NaN 66
Functions 66
Numeric input/output 67
Numerics environment 67
About the C SANE Library 67
Programming with IEEE arithmetic 68

3 The Standard C Library 69

Chapter organization 73
Diagnostics—Assert.h 74
Character handling—CType.h 75
Character testing—`isalpha`, `isupper`, `islower`, `isdigit`, `isxdigit`, `isalnum`, `isspace`, `ispunct`, `isprint`, `isgraph`, `iscntrl` 75
Character case—`toupper`, `tolower`, `_toupper`, `_tolower`, `toascii` 77
Errors—ErrNo.h 78
Localization—Locale.h 82
The `setlocale` function 82
The `localeconv` function 83
Mathematics—Math.h 87
Trigonometric functions—`sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2` 88
Hyperbolic functions—`sinh`, `cosh`, `tanh` 89
Exponential and logarithmic functions—`exp`, `log`, `log10`, `frexp`, `ldexp`, `modf` 89
Power functions—`pow`, `sqrt` 90
Nearest integer, absolute value, and remainder functions—`floor`, `ceil`, `fmod`, `fabs` 91
Euclidean distance function—`hypot` 92
Floating-point-to-string conversion functions—`ecvt`, `fcvt` 92
Non-local jumps—SetJmp.h 93
Signal handling—Signal.h 95
Specifying signal handling—the `signal` function 96
Sending a signal—the `raise` function 97
The Standard Input/Output package—StdIO.h 98
Operations on files 102
Removing a file—`remove` 102
Renaming a file—`rename` 102
Creating a temporary file—`tmpfile` 103
Naming a temporary file—`tmpnam` 103
File access functions 104

Closing a file—fclose, fflush 104
Opening a file—fopen, freopen, fdopen 105
File buffering—setbuf, setvbuf 106
Formatted input/output functions 108
Printing formatted output—printf, fprintf, sprintf, vprintf, vfprintf, vsprintf 108
Converting formatted input—scanf, fscanf, sscanf 112
Character input/output functions 117
Getting character input—getc, getchar, fgetc, getw 117
Getting string input—gets, fgets 118
Character output—putc, putchar, fputc, putw 119
String output—puts, fputs 120
Undoing an input—ungetc 120
Direct input/output—fread, fwrite 121
File positioning—fseek, rewind, ftell, fgetpos, fsetpos 122
Error-handling—feof, ferror, clearerr, perror, fileno 123
Input/output control (non-ANSI extension)—IOCtl.h 124
The ioctl function 124
Low-level file I/O (non-ANSI extensions)—FCntl.h 126
Creating a file—creat 127
Opening a file—open 127
Closing a file—close 129
Removing a file—unlink 129
Reading from a file—read 130
Writing to a file—write 131
File positioning—lseek 132
Get file control and status information—faccess 133
Duplicate an open file descriptor—dup 136
Duplicate a file descriptor—fcntl 136
Variable arguments—StdArg.h 138
General utilities—StdLib.h 139
String-conversion functions 140
String-to-int conversion—atoi, atol 140
String–to–long-conversion—strtol, strtoul 140
String–to–float conversion—atof 141
String to double conversion—strtod 142
Random number functions—rand, srand 142
Memory-management functions—malloc, free, realloc, calloc 143
Communication with the environment 144
Program termination—abort 144

String to double conversion—strtod 142
Random number functions—rand, srand 142
Memory-management functions—malloc, free, realloc, calloc 143
Communication with the environment 144
Program termination—abort 144
Install a function to be executed at program termination—atexit 145
Terminate the current application—exit 146
Access exported MPW shell variables—getenv 147
Communicate with the host system—system 148
Searching and sorting utilities 148
Search for an item in a sorted array—bsearch 148
Quicker sort—qsort 149
Integer arithmetic functions 150
Integer absolute value—abs, labs 150
Integer division—div, ldiv 150
Multibyte character functions—mblen, mbtowc, wctomb 151
Multibyte string functions—mbstowcs, wcstombs 152
String handling—String.h 153
String function conventions 154
Copying functions—memcpy, memccpy, memmove, strcpy, strncpy 154
Concatenation functions—strcat, strncat 155
Comparison functions—memcmp, strcmp, strncmp, strcoll, strxfrm 156
Search functions—memchr, strchr, strpbrk, strspn, strcspn, strstr, strtok 157
Miscellaneous functions—memset, strerror, strlen 159
Date and time—Time.h 159
Time-manipulation functions—clock, difftime, mktime, time 160
Time conversion functions—asctime, ctime, gmtime, localtime, strftime 161
Characteristics of floating types—Float.h 163
Sizes of integral types—Limits.h 164
Common definitions—StdDef.h 165

4 Using the Macintosh interfaces 167

About the Macintosh interfaces 169
Header files 169
How the interface is implemented 169
Parameter types 170
Passing string parameters 170
Correspondences between Pascal variant records and C structures 171
The `Str255` type 171
Functions using strings, points, and cells 172

A C Compiler Options 173

B Files Supplied With MPW 3.0 C 179

C Compiler (MPW C1:Tools folder) 181
Object files(MPW C1:Libraries:CLibraries folder) 181
Example source files(MPW C2:Examples:CExamples folder) 181
Library header files(MPW C2:Interfaces:CIncludes folder) 182

C Calling conventions and type correspondence 185

C-style calling conventions 187
C parameters 187
C function results 187
C register conventions 188
Pascal-style calling conventions 188
Pascal parameters 188
Pascal function results 189
Pascal register conventions 189
Correspondence between C and Pascal data types 189

D Differences between MPW 2.0 C and MPW 3.0 C 195

Language 197
- Preprocessor 197
- Keywords 197
- Character constants: the newline, carriage-return, and vertical-tab characters 197
- Conversions of integer types 197
- Bit-fields 198
- Pascal-style functions 198
- Direct functions 199
- Function prototypes 199
- Segmentation specifications 199
- Signed characters 200
- Structure results 201
- Structure comparison (here be dragons) 201
- Register use 202
- `switch` statements 203
- Language anachronisms 203
 - Assignment operators 203
 - Initialization 203
 - Structures and unions 204

C compiler options 204

New files 204

Standard C Library 205
- Header files 205
- Signal-handling mechanism 205
- Variable arguments 205
- Function `cfree` 205
- Function `scanf` and `printf` 206

Macintosh interfaces 206
- Pascal-style strings (`Str255`) 206
- Functions using strings, points, and cells 207
 - Using C strings with the Macintosh interfaces 208
- Converting MPW 2.0 C source code 209

Glossary 211

Index 223

Figures and tables

1 About MPW C 1

Table 1-1 Library object files 6
Table 1-2 Example source files used by MPW C 7
Table 1-3 68851 Header-file search rules 10
Table 1-4 Header-file search rules (partial pathnames) 11
Table 1-5 Code compiled to run on any model of Macintosh 16
Table 1-6 Code compiled to use the MC68881 on the Macintosh II 16

2 The MPW C Language 29

Table 2-1 Size and range of data types 43

3 The Standard C Library 69

Table 3-1 Character-testing macros 76
Table 3-2 I/O errors 79
Table 3-3 Setlocale categories 82
Table 3-4 Standard open files 100

4 Using the Macintosh interfaces 167

Table 4-1 Correspondences between variant records and structures 171

C Calling conventions and type correspondence 185

Table C-1 Parameter and result data types 190

Preface

THIS MANUAL DESCRIBES MACINTOSH PROGRAMMER'S WORKSHOP (MPW™) C, version 3.0. It tells you what you need to write applications, tools (programs that run under the MPW environment), desk accessories, and drivers for the Apple® Macintosh® computer. It assumes you already know the C programming language.

This manual includes information about compiler options, the MPW 3.0 C language itself, and the libraries supplied with MPW C. ■

Contents

What this manual contains xv
Other reference material you'll need xvi
Typographic and spelling conventions xvii
 Language notation xvii
 Boldface xviii
 Spelling and capitalization xviii
Aids to understanding xix
 For more information xix

What this manual contains

This manual contains these sections:

- This Preface describes the manual and directs you to other reference books with information about the C language and the Macintosh programming environment.
- Chapter 1, "About MPW C," introduces MPW C and the C libraries; tells you about the sample programs and tells how to build a program; describes the C (Compile) command and its options; tells you which include files to compile with and which library files to link with; describes the considerations for using MPW C to generate programs for different versions of the Macintosh (Macintosh Plus, Macintosh SE, and Macintosh II); and explains how to write applications, tools, and desk accessories.
- Chapter 2, "The MPW C Language," describes Apple extensions to C and the differences between this implementation of C and other implementations.
- Chapter 3, "The Standard C Library," documents a collection of basic routines that let you read and write files, examine and manipulate strings, perform data conversion, acquire and release memory, and perform mathematical operations.
- Chapter 4, "Using the Macintosh Interfaces," discusses the interfaces between MPW C and the Macintosh ROM and RAM routines. These interfaces enable you to write C programs that access the routines described in *Inside Macintosh,* Volumes I through V.
- Appendix A, "C Compiler Options," explains the syntax and options of the C command.
- Appendix B, "Files Supplied With MPW C," contains a list of all the files that are supplied with this product.
- Appendix C, "Calling Conventions and Type Correspondences," defines the conventions for calling C and Pascal routines. It explains how parameters are passed to functions, how function results are returned, and how registers are saved and restored.
- Appendix D, "Differences Between MPW 2.0 C and MPW 3.0 C," describes changes to MPW C in language definitions, interfaces, and so forth, and tells how to convert code for MPW 3.0 C.

Other reference material you'll need

You'll need to be familiar with these additional reference materials:

- *Macintosh Programmer's Workshop 3.0 Reference.* (APDA, 1988). Describes the Macintosh Programmer's Workshop environment in which the C compiler operates, including the editor, linker, and other important tools.
- *The C Programming Language.* (First edition. Brian W. Kernighan and Dennis M. Ritchie. Prentice-Hall, 1978). The standard reference book for the C language as originally defined. (This book is herein called *K&R 1*, or simply *K&R*. The language it defines is called *K&R C.*)
- *Inside Macintosh* , Volumes I-III. (Addison-Wesley, 1985). Contains information you need in order to program with the Macintosh ROM and associated RAM routines: these routines provide windows, alert boxes, menus, graphics, and much more. Volumes I through III apply to all Macintosh computers.
- *Inside Macintosh* , Volume IV. (Addison-Wesley, 1986). Describes the 128K ROM routines available with the Macintosh Plus or Macintosh 512K enhanced, as well as the Macintosh II and Macintosh SE. (Some of these routines, such as the hierarchical-file-system routines, are also available on disk via the System file for machines that have the 64K ROM.)
- *Inside Macintosh*, Volume V. (Addison-Wesley, 1987). Describes the 256K ROM routines available with the Macintosh II and Macintosh SE. (Some of these routines, such as the TextEdit functions, are also available on the Macintosh Plus.)
- *MacsBug Reference.* (APDA, 1988). Describes MacsBug, Apple's object-level debugger.
- *SADE Reference.* (APDA, 1988). Describes SADE™, Apple's source-level debugger.

You may also want to be familiar with these additional reference materials:

- *ResEdit Reference.* (APDA, 1988.) Describes ResEdit, Apple's resource editor.
- *The C Programming Language.* (Second edition. Brian W. Kernighan and Dennis M. Ritchie. Prentice-Hall, 1988.) The standard reference book for the C language, rewritten for draft proposed ANSI C. (This book is herein called *K&R 2.*)
- *C: A Reference Manual.* (Second edition. Samuel P. Harbison and Guy L. Steele, Jr. Prentice-Hall, 1987.) A standard reference book for the C language with the AT&T extensions used in most UNIX® operating system environments. The second edition contains a chapter on draft proposed ANSI C.
- *Draft Proposed American National Standard for Information Systems—Programming Language C.* (ANSI document X3J11/88-002, May 13, 1988.) This standard and its accompanying rationale define ANSI C, codifying many of the extensions added since 1978. This book is herein referred to as the ANSI draft C standard. The language it describes is referred to as draft proposed ANSI C.

- *Rationale for Draft Proposed American National Standard for Information Systems—Programming Language C.* (ANSI Document Number X3J11/86-152, October 1, 1986.) The rationale behind the ANSI draft C standard.
- *Notes on the Draft C Standard.* (Thomas Plum. Plum Hall Inc., 1987.) Additional explanation of the ANSI draft C standard, by the Vice Chair of the ANSI X3J11 committee on the standardization of the C language.
- *Apple Numerics Manual.* Second edition. (Addison-Wesley, 1988.) For the programmer who wants more understanding or control of the underlying floating-point arithmetic in MPW C. It describes the Standard Apple Numerics Environment (SANE®), which includes extended-precision floating-point arithmetic as specified by IEEE Standard 754. It describes each routine in detail, including exception handling, and explains how to control the floating-point environment.
- *Motorola MC68020 32-Bit Microprocessor User's Manual.* (Second edition. Prentice-Hall, 1985.) Describes the MC68020 processor in detail for hardware and software engineers.
- *MC68881 Floating-Point Coprocessor User's Manual.* (Motorola, Inc., 1985.) Describes the instruction set and addressing conventions used by the MC68881 floating-point coprocessor, which is used in the Macintosh II.

Typographic and spelling conventions

This section describes the conventions used in this manual.

Language notation

This manual uses certain conventions in common with most books on C.

- C code is in a monospaced font:

```
int ndigit[10]
```

- Replaceable items in syntax descriptions are in italics:

```
else if (condition )
    statement
```

Here *condition* and *statement* are expressions that are replaced by actual C expressions. The `else if` and the parentheses are C code.

Boldface

Terms that appear in boldface type when first defined in the text also appear in the glossary. For example, "An **application** is a program that can be run under the Macintosh MultiFinder™."

Spelling and capitalization

In the Standard C Library, the spelling and capitalization of identifiers are generally as specified in the synopsis for each Standard C Library routine. Most function and parameter names are spelled entirely in lowercase. Most constant values are spelled entirely in uppercase.

In the *Inside Macintosh* Libraries, the spelling and capitalization of identifiers are exactly as specified in *Inside Macintosh.* Constants, variables, parameter names, fields within structures, and enumerated-type elements begin with a lowercase letter. Routines and data types begin with an uppercase letter. Within an identifier, letters that begin new words in English are capitalized. All other letters are lowercase. When a name contains an acronym, the case of the entire acronym is determined by the case of its first letter (for example, `GetOSEvent` and `teJustLeft`).

An alternative set of interface routines, spelled in lowercase, is included for compatibility with some existing code. These routines are described in the section "Functions Using Strings, Points, and Cells," in Appendix D.

The SANE®interface follows standard C spelling conventions, not *Inside Macintosh* conventions.

Aids to understanding

Look for these visual cues throughout the manual:

▲ **Warning** Warnings like this indicate potential problems. ▲

△ **Important** Text set off in this manner presents important information. △

◆ *Note:* Text set off in this manner presents notes, reminders, and hints.

For more information

APDA™ provides a wide range of technical products and documentation, from Apple and other suppliers, for programmers and developers who work on Apple equipment. (MPW is distributed through APDA.) For information about APDA, contact the

APDA
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 33-G
Cupertino, CA 95014-6299

1-800-282-APDA, or 1-800-282-2732
Fax: 408-562-3971
Telex: 171-576
AppleLink: DEV.CHANNELS

If you plan to develop hardware or software products for sale through retail channels, you can get valuable support from Apple Developer Programs. Write to

Apple Developer Programs
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 51-W
Cupertino, CA 95014-6299

Chapter 1 About MPW C

MPW™ C IS A COMPLETE IMPLEMENTATION OF THE C PROGRAMMING LANGUAGE. It consists of the C compiler, the Standard C Library, the Macintosh® interfaces, the C SANE® Library, and example programs. The C compiler is based on Kernighan and Ritchie's original definition of C (**K&R C**), with a number of extensions from the ANSI draft C standard, as well as some Apple extensions. The Standard C Library is based on the standard C library described in the ANSI draft C standard. The Standard C Library also provides the Integrated Environment used by MPW tools. (Tools are described later in this chapter; the Integrated Environment is described in the *MPW 3.0 Reference.*) The Macintosh interfaces provide access from C programs to the routines described in *Inside Macintosh,* as well as to the Graf3D Library. The C SANE Library supports the Standard Apple Numerics Environment described in the *Apple Numerics Manual* and in the *MPW 3.0 Reference.* ■

Contents

About the C compiler 3
About MPW C version 3.0 3
 Floating-point enhancements 4
 MC68020 enhancements 4
 The Macintosh interfaces 4
About the C libraries 5
About the C examples 7
Installing MPW C 9
Using header files 9
Segmentation control 11
Creating resources 13
Creating an application in C 13
 Building an Application 13
 Compiling an application 14
 Linking an application 15
Creating an MPW tool in C 17
 Building a tool 18
 Compiling a tool 18

Linking a tool 19
Creating a desk accessory in C 19
Desk accessory restrictions 19
The DRVRRuntime library 20
Desk accessory routines 21
Building a desk accessory 21
Compiling a desk accessory 22
Linking a desk accessory 22
Stand-alone code resources 23
Creating code for different models of Macintosh 24
Source code 24
Header files 24
Compiler options 25
Creating code for the MC68881 and MC68020 25
MC68881 enhancements 25
Linking with the `-mc68881` option 26
The `extended` data format 27
MC68881 instructions 27
Floating-point registers 27
MC68020 enhancements 27

About the C compiler

The C Programming Language (first edition) by Kernighan and Ritchie provides an authoritative definition of C by its creators: this original version of C is called K&R C. However, Kernighan and Ritchie left many details to implementors, and many extensions have been added to the language since the original publication of their book.

Recently, an ANSI committee has been developing a standard for the C language. This standard includes many of the common extensions in a uniform way. In addition to a number of other improvements, the ANSI draft C standard provides function prototyping, which brings strong type checking to C; the `signed` keyword, which allows certain types to be unsigned by default; types `enum`, `void`, and `void *` (pointer to `void`); and the use of structures as function parameters and function results.

MPW C also contains these Apple® extensions to the C language:

- the `pascal` type qualifier, which allows calls to and from Pascal programs and the Macintosh Toolbox
- the arithmetic data types `single` (synonym for `float`), `extended` (synonym for `long double`), and `comp`, required by the Standard Apple Numerics Environment (SANE)

Both the ANSI C extensions and the Apple extensions in MPW C are described in Chapter 2.

About MPW C version 3.0

Version 3.0 of MPW C is a replacement for version 2.0 of MPW C. It is largely compatible with version 2.0, but there are some incompatibilities that require source-code changes. (A utility program, CCvt, is provided to assist in making these changes: see Appendix D.) Version 3.0 provides a number of ANSI-C features, plus some Apple-specific extensions.

The compiler incorporates a number of the most important features of draft proposed ANSI C, as defined in the *American National Standard for Information Systems—Programming Language C*. This standard is currently in draft form—it is herein referred to as the ANSI draft C standard—and is expected to be approved during 1989.

ANSI C header files and language libraries are included with the MPW 3.0 compiler. These provide all the functions specified by the ANSI draft C standard, as well as some Apple extensions.

Appendix D, "Differences Between MPW 2.0 C and MPW 3.0 C," explains changes you may need to make to your code for compatibility with MPW 3.0 C and tells how to use the CCvt program to help convert your code.

Floating-point enhancements

Floating-point operations in any program running on the Macintosh II gain some speed from the **MC68881** coprocessor because the standard SANE libraries use it to increase the speed of calculations. For applications that need as much speed as possible and can be restricted to run only on the Macintosh II, another level of speed improvement is possible by direct use of this coprocessor through the `-mc68881` and `-elems881` options. These options are described later in this chapter.

MC68020 enhancements

The MPW 3.0 C Compiler provides direct support of the MC68020 through the `-mc68020` option, which causes the compiler to use the MC68020 instruction set, including multiplications of `int`s, division of `int`s, and bit-field operations. Use this option for applications that need as much speed as possible and run only on machines, like the Macintosh II, that have an MC68020 processor. This option is described later in this chapter.

The Macintosh interfaces

A set of header files and libraries called the Macintosh interfaces is provided with MPW C and gives you access from C to the User Interface Toolbox and Macintosh Operating System routines built into the Macintosh ROMs. The Pascal versions of these routines are described in detail in *Inside Macintosh,* Volumes I through V. Volume V describes the ROM routines introduced with the Macintosh SE and the Macintosh II family.

Chapter 4 in this manual describes the Macintosh interfaces. Some of the interfaces, such as the Color Manager and the Slot Manager, are useful only on a Macintosh that supports a color display. The new functions for font control in dialogs, supplied with the TextEdit interface, apply to all models of Macintosh that use the current system disk.

See "Creating Code for Different Models" later in this chapter for procedures for writing and compiling code to run on different systems.

About the C libraries

MPW C includes the following libraries:

- The **Standard C Library** is a standardized collection of basic routines that let you read and write files, examine and manipulate strings, perform data conversion, acquire and release memory, and perform some mathematical operations. These functions are portable between MPW C and ANSI C. This library also contains functions that support MPW tools. Chapter 3 describes these routines in detail.
- The **Macintosh interfaces** are a set of interfaces between C and the Macintosh ROM and RAM routines. These interfaces enable you to write C programs that access the routines described in *Inside Macintosh,* as well as the Graf3D routines. Chapter 4 tells how to use these interfaces.
- The **C SANE Library** provides mathematical functions and supports floating-point arithmetic. Some of these routines are called through the Standard C Library: they are documented in Chapter 3. The other SANE routines are called through the Macintosh interfaces. The semantics of these routines is described in detail in the *Apple Numerics Manual.*

Within Chapter 3, the material is alphabetical by function or library name. All of the identifiers defined in the libraries are listed in a library index in Appendix C.

Table 1-1 lists the library object files used by MPW C. The first five files, provided with the Macintosh Programmer's Workshop, are shared with other languages and appear in the {Libraries} directory. The remaining files, provided with MPW C, are used only with C and appear in the {CLibraries} directory.

Three of the C libraries (CLib881.o, CSANELib881.o, and Math881.o) are for use only with the MC68881 floating-point coprocessor on the Macintosh II.

■ **Table 1-1** Library object files used by MPW C

Libraries shared with other languages—{Libraries} directory

File	Description
DRVRRuntime.o	Run-time support for desk accessories and other drivers.
HyperXLib.o	HyperCard `'XCMD'` routines.
Interface.o	Macintosh interface routines shared with other languages.
PerformLib.o	Performance measurement routines.
Stubs.o	Stubs used by the linker to replace unused library routines. Used in linking MPW tools to remove support for standalone text I/O.
ToolLibs.o	Routines normally used by tools, including the spinning cursor and error manager.

C Libraries—{CLibraries} directory

File	Description
CInterface.o	Macintosh interface routines specifically for C (lowercase forms).
CLib881.o	Standard C Library overrides, using the 96-bit `extended` format for compatibility with the MC68881. If you use the `-mc68881` option, put CLib881.o before the other libraries in {CLibraries} in the link list.
Complex.o	Complex SANE numerics for C.
Complex881.o	Complex SANE numerics for C, using the MC68881 for some operations. If you use the `-mc68881` option, link with this file instead of with Complex.o.
CRuntime.o	Execution starting point for applications and tools, data initialization, QuickDraw data, low-level I/O, signal handling, and built-in routines.
CSANELib.o	SANE numerics for C, using only the software SANE routines.
CSANELib881.o	SANE numerics for C, using the MC68881 for some operations. If you use the `-mc68881` option, link with this file instead of with CSANELib.o.
Math.o	Math functions from the Standard C Library, including conversions, exponential and logarithmic functions, trigonometric functions, and hyperbolic functions.
Math881.o	Math functions from the Standard C Library, using the MC68881 for floating-point operations. If you use the `-mc68881` option, link with this file instead of with Math.o.
StdCLib.o	The majority of the Standard C Library routines, except for the mathematical functions in Math.o.

See "Linking an Application," later in this chapter, for more information on using these libraries.

About the C examples

The CExamples folder contains source files for sample applications, a sample MPW tool, a sample desk accessory, and a sample cdev, all written in C, plus a sample program that uses the C Performance Tools. Table 1-2 lists these files.

■ **Table 1-2** Example source files used by MPW C

Source files—{CExamples} directory

File	Description
Count.c	C source file for Count, a sample MPW tool supplied with MPW and documented in the *Macintosh Programmer's Workshop 3.0 Reference,* Chapter 2.
Count.r	Rez source file containing the Commando resource for the Count tool.
EditCDev.c	C source file for EditCDev, a sample control device (cdev) with a TextEdit item.
EditCDev.make	Makefile for the EditCDev application.
EditCDev.r	Rez source file for the EditCDev application.
FStubs.c	Stubs for those floating-point library routines not needed by MPW tools. The build instructions for Count.c (in the file MakeFile) use these stubs to reduce the size of the final tool. See "Linking a Tool," later in this chapter, for more information.
Instructions	Instructions for building each of the MPW 3.0 C sample programs. After installing MPW and MPW C (as described in the next section), open this file and follow the instructions.
MakeFile	Makefile for building several of the sample programs in the CExamples folder. (The other programs have their own makefiles.)

(continued)

■ **Table 1-2** (continued) Example source files used by MPW C

Source files—{CExamples} directory

File	Description
Memory.c	C source file for Memory, a sample desk accessory that displays the current free space in the application and system heaps, the free space on the default volume, and the name of the default volume. This information is updated every five seconds. When Memory is first opened, it calls `_MaxMem` to purge memory, thus showing the upper bounds on free space in the heaps.
Memory.r	Rez source file containing resources for the Memory desk accessory.
Sample.c	C source file for Sample, a minimal MultiFinder-aware sample application that is an embellished version of the sample application described in "A Simple Example Program" in Chapter 1 of *Inside Macintosh,* Volume 1.
Sample.h	Header file for the Sample application.
Sample.make	Makefile for the Sample application.
Sample.r	Rez source file containing resources (windows, menus, dialogs, *etc.*) for the Sample application.
SillyBalls.c	C source file for SillyBalls, a Color QuickDraw sample application.
SillyBalls.make	Makefile for the SillyBalls application.
TESample.c	C source file for TESample, a MultiFinder-aware TextEdit application.
TESample.h	Header file for the TESample application.
TESample.make	Makefile for the TESample application.
TESample.r	Rez source file for the TESample application.
TESampleGlue.a	Assembly-language source file for the TESample application.
TESampleGlue.a.o	Assembly-language object file for the TESample application (for MPW C users who don't have the MPW Assembler).
TestPerf.c	C source file for TestPerf, a sample MPW tool that uses the C Performance Tools.
TubeTest.c	C source file for TubeTest, a sample application using Color QuickDraw and Palette Manager.
TubeTest.make	Makefile for the TubeTest sample application.
TubeTest.r	Rez source file for the TubeTest sample application.

Installing MPW C

Instructions for installing MPW on a hard disk appear in Chapter 2 of the *Macintosh Programmer's Workshop 3.0 Reference*. When installing MPW by using the Install script, simply install the two MPW C disks at the same time.

Alternatively, you can install C with these steps:

1. Copy the file MPW C1:Tools:C (the C compiler) to the {MPW}Tools folder.
2. Copy the folder MPW C1:Libraries:CLibraries to the {MPW}Libraries folder.
3. Copy the folder MPW C2:Examples:CExamples to the {MPW}Examples folder.
4. Copy the folder MPW C2:Interfaces:CIncludes to the {MPW}Interfaces folder.

◆ *Note:* You can put the compiler, includes, and libraries in different directories, provided you change the default values of various shell variables defined in the Startup file. You can modify the file Startup itself or, preferably, modify the file UserStartup. The following variables determine the locations of files supplied with MPW C:

{Commands}	A comma-separated list of directories containing tools and applications. The directory containing the C compiler should appear in this list.
{CIncludes}	A comma-separated list of directories to search for C include files. This list should include the CIncludes directory.
{CLibraries}	The directory containing C library files. This variable should contain the pathname of the CLibraries directory.

For more information, see the *Macintosh Programmer's Workshop 3.0 Reference*.

Using header files

Header files (also called *include files*) are provided for both the Standard C Library and the Macintosh interfaces. These files are in the {CIncludes} directory. You can determine what header files to include for a specific library function or data structure by checking the appropriate section in Chapters 3 and 4.

The compiler searches several directories for header files, until the specified file is found. The compiler searches the directory containing the current input file, directories specified by using the `-i` option to the compiler, and directories specified in the shell variable {CIncludes}.

You specify the header files needed for your programs by using the `#include` statement, enclosing them in either double quotation marks or angle brackets:

`#include "`*filename*`"`
`#include <`*filename*`>`

By convention, double quotation marks are used for header files you create, and angle brackets for header files supplied with MPW.

```
#include "MyFile.h"
#include <Types.h>
```

The search rules the compiler uses in looking for the header file differ slightly in the two cases.

The form of the pathname also determines where the compiler looks for the header file. If a *full pathname* is specified, the compiler uses exactly that name and performs no search. A full pathname contains at least one colon (:), but doesn't begin with a colon. If a *partial pathname* is specified, the compiler searches several directories for the file. Partial pathnames either begin with a colon or don't contain any colons.

Header files can be nested up to 15 levels deep.

Tables 1-3 and 1-4 summarize the compiler's header-file search rules.

■ **Table 1-3** **68851** Header-file search rules (full pathnames)

Pathname	Description
`#include "`*volname*`:`*dir1*`:...:`*filename*`"`	Use the name as specified.
`#include <`*volname*`:`*dir1*`:...:`*filename*`>`	Use the name as specified.

■ **Table 1-4** Header-file search rules (partial pathnames)

Pathname	Description
`#include "`*filename*`"`	Search the following directories, in this order: 1. The directory of the source file that contains the `#include` statement. 2. Directories specified by the compiler's `-i` option, in the order specified. 3. Directories specified by the shell variable {CIncludes}.
`#include <`*filename*`>`	Search only the directories listed above in steps 2 and 3.

Segmentation control

A **segment** is a part of code that can be separately loaded into memory. Your program can be written without explicit segmentation, or it can contain a number of different segments.

Each `'CODE'` resource in the application's resource fork corresponds to a segment containing one or more functions. (The `'CODE'` resource with ID 0 contains the jump table; other `'CODE'` resources contain functions.) At run time, a segment is automatically loaded by the Segment Loader when you call one of the functions contained in the segment. The segment is not unloaded until the application explicitly unloads it by calling `UnloadSeg`. See *Inside Macintosh* for more information about the Segment Loader.

There are two main ways to specify which functions are placed in which segments. This section tells how to use the `#pragma segment` directive to specify segmentation. The *Macintosh Programmer's Workshop 3.0 Reference* explains how to use the Link command to modify a program's segmentation.

Segmentation helps you reduce your program's run-time memory requirements. A typical segmentation scheme divides a program into an initialization segment and a main processing segment. You can also put routines that are seldom executed—printing routines, for instance—in a separate segment that is not loaded when the program begins executing. This type of segmentation causes the program to be loaded faster, because the printing routines are not loaded until they are needed. If you don't specify segmentation, the compiler puts the entire program into a segment called **Main.** You can override the default name with the `-s` compiler option or the `#pragma segment` directive.

The `#pragma segment` directive also lets you specify several segments within a single source file. To assign source code to a segment, precede the code with a statement of the form

`#pragma segment` *segment-name*

The code following this statement is placed in the named segment until the compiler reads another `#pragma segment` directive or the end of the source file.

◆ *Note:* In a `#pragma` directive, segment names are case sensitive. Leading and trailing spaces are significant. Unless you want the segment name to start or end with spaces, leave exactly one space between `#pragma segment` and *segment-name* and no spaces after *segment-name*.

Code for a given segment does not have to be contiguous within the source file. The program may take the form

```
#pragma segment SegA
  ... function ...
#pragma segment SegB
  ... function ...
  ... function ...
#pragma segment SegA
  ... function ...
and so on.
```

The compiler marks each function with the name of its segment. Then the linker collects functions for a segment from various input files and places them in a single segment in the output file.

△ **Important** The `#pragma segment` directive will be supported in future releases of MPW C. An older form of segment specification, the `#define __SEG__` directive, is retained in the interest of compatibility with existing code: it will not be supported in future releases. We recommend that you replace any instances of it with the `#pragma segment` directive. △

Creating resources

Noncode resources, such as the resources that specify menus, windows, and dialogs, can be created by using the resource editor (ResEdit) and the Resource Compiler (Rez). These tools are described in the *Macintosh Programmer's Workshop 3.0 Reference* and the *ResEdit Reference.* The makefiles for the example programs show how to incorporate the resource compiler in the building of an application or desk accessory.

Creating an application in C

An **application** is a program that can be launched by the Macintosh Finder™ program. Applications can also be launched by the MPW Shell: execution of the MPW Shell is suspended, and the application takes over the computer's memory and display while executing. If the shell is running under MultiFinder, the application is launched in its own partition.

The code for an application is contained in `'CODE'` resources in the resource fork of its file. Additional resources in the same file describe the menus, windows, dialogs, strings, and other resources used by the application. *Inside Macintosh* explains in detail how to write a Macintosh application.

This section outlines the steps for building an application in MPW C. The file Instructions in the CExamples folder describes some of the tools that can be used to automate the process. The file MakeFile in the CExamples folder illustrates the use of some of the tools. The *Macintosh Programmer's Workshop 3.0 Reference* describes these tools in detail.

Building an Application

The easiest way to build any MPW program is to use the commands found in the Build menu. We will build Sample, an application from the Examples folder. The source files for Sample are Sample.c, Sample.h, Sample.make, and Sample.r; since a makefile already exists, you don't need to create one. (The *MPW Reference* explains how to create a makefile by using the CreateMake script.) Using the Directory menu, set the current directory to *diskname:*MPW:Examples:CExamples:, where *diskname* is the name of your disk.

Now select Build from the Build menu and type the program name `sample`. You will see something like the following code on the screen. (A few line breaks and continuation characters have been added for readability.)

```
# 3:32:38 PM ----- Build of sample.
# 3:32:38 PM ----- Analyzing dependencies.
# 3:32:42 PM ----- Executing build commands.
    C Sample.c -o Sample.c.o -d MPW3 -r
    Link -o Sample Sample.c.o ∂
        "diskname:MPW:Libraries:CLibraries:"CRuntime.o ∂
        "diskname:MPW:Libraries:CLibraries:"CInterface.o ∂
        "diskname:MPW:Libraries:Libraries:"Interface.o
    SetFile Sample -t APPL -c 'MOOS' -a B
    Rez -rd -o Sample Sample.r -append
# 3:33:39 PM ----- Done.
  sample|
```

Press Enter to launch the sample application. Quit (Command-Q) returns you to the MPW Shell.

The next two sections explain how to build an application step by step without using the Build menu.

Compiling an application

To compile a C program, first start the MPW Shell application, then execute the `C` command from any window. Typically the command specifies options and the name of the source file to the compiler, although neither is required. For example, the command

```
C  -p  Sample.c
```

compiles the source file Sample.c, producing the object file Sample.c.o. The `-p` option specifies that progress information should be written to diagnostic output. This information appears in the window.

If you enter the command

```
C
```

(that is, the C command without a filename), the compiler reads from **standard input.** This means that the compiler reads any text that you subsequently enter. This feature allows you to run the compiler interactively. You can tell that the compiler (rather than the MPW Shell) is reading the text you enter, because the name *C* appears in the status box in the lower-left corner of the active window. Once the compiler is running interactively, you can enter source code in any window, composing your program as you go. Press the Enter key after each line of C code to send the code to the compiler. To terminate input, press the Command and Enter keys simultaneously. When the compiler compiles standard input, it creates an object file named C.o.

- *Note:* It can be very confusing if you accidentally run the compiler interactively. If you get the impression that the MPW Shell isn't listening to the text you enter, check the status box in the lower-left corner of the active window. It contains either the name of the MPW Shell (if the shell is listening) or the name of the command that is currently executing.

A complete specification of the C command—including input, output, and diagnostic specifications, status values, and options—appears in Appendix A and in the *Macintosh Programmer's Workshop 3.0 Reference.*

Linking an application

The linker is used to combine object files from several separate compilations, together with any necessary library object files, to produce the executable code resources for a program. The linker either creates a new resource file, containing only the code resources for your program, or replaces the code resources in an existing resource file, leaving other resources, such as menus and dialogs, intact. This allows you to run the Resource Compiler either before or after running the linker. The linker is described in detail in Chapter 10 of the *Macintosh Programmer's Workshop 3.0 Reference.*

An application written partly or totally in C should be linked with the libraries listed in Tables 1-5 and 1-6.

■ **Table 1-5** Code compiled to run on any model of Macintosh

Macintosh interfaces	Run-time support	Standard C Library
{Libraries}Interface.o	{CLibraries}CRuntime.o	{CLibraries}StdCLib.o {CLibraries}CInterface.o {CLibraries}CSANELib.o {CLibraries}Math.o {CLibraries}Complex.o

■ **Table 1-6** Code compiled to use the MC68881 on the Macintosh II

Macintosh interfaces	Run-time support	Standard C Library
{Libraries}Interface.o	{CLibraries}CRuntime.o	{CLibraries}CLib881.o {CLibraries}CInterface.o {CLibraries}StdCLib.o {CLibraries}CSANELib881.o {CLibraries}Math881.o {CLibraries}Complex881.o

If you specify unnecessary files in the Link command, the linker displays a message listing which files can be removed from your build instructions.

If you are using the `-mc68881` compiler option, you must place the file CLib881.o before the other libraries in {CLibraries} in the link list. This file contains some definitions that override 80-bit versions in other libraries. The linker uses the first definition it reaches, then displays warning messages when it encounters duplicate definitions. You can use the `-d` linker option to suppress warnings about duplicate definitions.

Programs written partly in C and partly in assembly language or Pascal should be linked with the file CRuntime.o and not the file Runtime.o. The linker will detect several duplicate entry points when linking with both the Pascal and the C libraries. All but one of these duplicates can be safely ignored, because the copies of the routines are identical.

If execution is expected to begin with the C function `main`, no special precautions are necessary. However, if your main program is written in assembly language or Pascal but parts of your program are written in C (and must therefore be linked with the file CRuntime.o), the object file containing your main program must appear before CRuntime.o in the list of object files passed to the linker.

A C program that calls a Pascal function or procedure requires an external `pascal` declaration. (See "Pascal-Style Functions" in Chapter 2.)

Creating an MPW tool in C

A **tool** is a program that operates within the MPW Shell environment. The C compiler, Rez, and Link are all tools. You can write your own tools in C, Pascal, or assembly language. Chapter 12 of the *Macintosh Programmer's Workshop 3.0 Reference* describes tools and how they are created. This section contains specific information about writing tools in C.

You execute a tool by entering an MPW command. The parameters specified in the command line are passed as parameters to the function `main`. The shell variables that are exported are also passed as a parameter to `main`; they can be accessed directly or by using the `getenv` function from the Standard C Library. To access these parameters, begin the function `main` with the following parameter list:

```
main(
   int argc,       /* number of arguments */
   char *argv[],   /* pointer to array of argument strings */
   char *envp[]    /* pointer to array of variable definitions */
   )
```

Additional details about parameters to tools may be found in Chapter 12 of the *Macintosh Programmer's Workshop 3.0 Reference.*

Tools have direct access to MPW Shell windows and selections. The `FILE` variables `stdin`, `stdout`, and `stderr` refer to MPW's standard input, standard output, and diagnostic output, respectively. By default, Standard C Library I/O functions read standard input (text entered from the shell) and write to standard output. Any files opened by tools, using either Standard C Library functions or *Inside Macintosh* library functions, will read and write to windows if the file specified is open in a window. The contents of the window are read or written in place of the data fork of the file. Selections in windows can also be read and written as if they were files, by adding the suffix `.§` to the filename: for example, *diskname*:MPW:Worksheet.§. (The § character is produced by the keystroke Option-6.).

Building a tool

The easiest way to build any program in MPW is to use the Build menu. We will build Count, a tool that counts characters in files. The source files for Count are Count.c, FStubs.c, Count.r and MakeFile; since a makefile already exists, you don't need to create one. Using the Directory menu, set the current directory to *diskname*:MPW:Examples:CExamples:, where *diskname* is the name of your disk.

Now select Build from the Build menu and type the program name `Count`. You will see something like the following code on the screen. (A few line breaks and continuation characters have been added for readability.)

```
# 3:08:36 PM ----- Build of Count.
# 3:08:36 PM ----- Analyzing dependencies.
# 3:08:39 PM ----- Executing build commands.
    Rez Count.r -o Count -append
    Link -w -c 'MPS ' -t MPST Count.c.o FStubs.c.o -sn ∂
        STDIO=Main -sn INTENV=Main -sn %A5Init=Main ∂
        "diskname:MPW:Libraries:"Stubs.o ∂
        "diskname:MPW:CLibraries:"CRuntime.o ∂
        "diskname:MPW:CLibraries:"StdCLib.o ∂
        "diskname:MPW:CLibraries:"CInterface.o ∂
        "diskname:MPW:Libraries:"Interface.o ∂
        "diskname:MPW:Libraries:"ToolLibs.o -o Count
# 3:08:51 PM ----- Done.
Count|
```

The insertion point follows the word `Count`, awaiting your input. Now type

```
-c Count.c
```

and press Enter. You will get a count of the number of characters in the source file Count.c:

```
   3684
```

The next two sections explain how to build a tool step by step without a Build menu.

Compiling a tool

You compile a tool in exactly the same way you compile an application. The information above regarding include-file search rules, segmentation, and resources applies equally to tools and applications.

Linking a tool

The MPW Shell recognizes a tool by the type and creator. Specify the following options when linking a tool:

```
Link -t MPST -c "MPS " ...
```

This command specifies the file type and creator of an MPW tool. Follow the same library linking rules for tools as for applications. (See "Linking an Application," earlier in this chapter.) In addition, if your tool calls any of the spinning cursor or error manager routines, link with the following libraries:

```
{Libraries}Stubs.o
{Libraries}ToolLibs.o
```

The file Stubs.o contains a collection of "stubs," or dummy routines, for several functions that are defined in the run-time library but are not necessary for MPW tools running under the MPW Shell. You can use these stubs to reduce the size of a tool. Stubs.o should be linked in before any of the other libraries. It is usually wise to link with this file. See the Count.c build instructions in the file MakeFile.

Use Appendix C to determine which file to include when the linker gives an unresolved reference.

Creating a desk accessory in C

A **desk accessory** is a program that you run by selecting it from the Apple menu. It shares its execution environment with the currently executing application. Information on writing desk accessories appears in the Desk Manager and Device Manager chapters of *Inside Macintosh* and in the *Macintosh Programmer's Workshop 3.0 Reference*. This section contains information specific to writing desk accessories in MPW C.

Desk accessory restrictions

A desk accessory has neither a jump table nor a global data area. The following restrictions apply:

- Because it does not have a jump table, a desk accessory must be in a single segment. Either omit segmentation specifications so that all your code is placed in the default Main segment, or use identical segmentation specifications for all of your functions. Use the `Link` command to move any library routines you use into your single segment.
- Because it does not have a global data area, a desk accessory written in C may not use `extern` or `static` variables. Furthermore, a desk accessory cannot call library routines that require global data. You can use literal strings as long as they are not stored in a global data area: use the `-b` option to keep them within the desk accessory's code segment.

Programming hints for avoiding these restrictions appear in Chapter 8 of the *Macintosh Programmer's Workshop 3.0 Reference.*

The DRVRRuntime library

Desk accessories have traditionally been written in assembly-language source code, partly because of the peculiar resource format used by the system for desk accessories, the `'DRVR'` resource. Setting up the `'DRVR'` layout header, passing register-based procedure parameters, and coping with the nonstandard exit conventions of the driver routines have made it fairly difficult in the past for programmers not familiar with assembly language to implement desk accessories in higher-level languages.

To overcome these difficulties and simplify the task of writing a desk accessory in C, MPW provides the library DRVRRuntime.o and the resource type `'DRVW'` declared in MPWTypes.r. Together they compose the driver layout header and the five entry points that set up the `open`, `prime`, `status`, `control`, and `close` functions of a driver. For more information about `'DRVR'` resources, see the Device Driver chapter of *Inside Macintosh,* Volume II. For an example defining desk accessory resources, see the file Memory.r in the folder CExamples.

Using the library DRVRRuntime.o to create desk accessories offers a number of advantages:

- No assembly-language source is required. Each of the driver routines—`DRVROpen`, `DRVRPrime`, `DRVRStatus`, `DRVRControl`, and `DRVRClose`—can be written in C by using standard calling conventions.
- The DRVRRuntime library handles desk accessory exit conventions: your routines simply return a result code.

The DRVRRuntime library consists of a main entry point that overrides the C run-time main entry point. The DRVRRuntime entry point contains driver "glue" that sets up the parameters for you, calls your routine, and performs the special exit code required by a desk accessory to return control to the system. Your routines perform the actions of the desk accessory, such as opening a window or responding to mouse clicks in it.

Desk accessory routines

Desk accessories that use the library DRVRRuntime must contain the five functions `DRVROpen`, `DRVRPrime`, `DRVRStatus`, `DRVRControl`, and `DRVRClose`. All of these functions have the same parameter and result types. They are declared as Pascal-compatible functions so that the library DRVRRuntime can be used for writing desk accessories in C, Pascal, and assembly language. Each of these five routines should be declared as follows:

```
pascal OSErr DRVROpen(ctlPB,dCtl)
    CntrlParam *ctlPB;
    DCtlPtr dCtl;
{
    ... your code ...
    return(resultCode);
}
```

Types `CntrlParam` and `DCtlPtr` are defined in the file Files.h. Type `OSErr` is a `short` and is defined in Types.h. Details on each function appear in Chapter 8 of the *Macintosh Programmer's Workshop 3.0 Reference,* in the Desk Manager chapter of *Inside Macintosh,* Volume I, and in the Device Manager chapter of *Inside Macintosh,* Volume II.

Building a desk accessory

The easiest way to build any program in MPW is to use the Build menu. We will build Memory, a sample desk accessory that displays the memory available in the application and system heaps, and on the boot disk.

The source files for Memory are Memory.c, Memory.r, and MakeFile; since a makefile already exists, you don't need to create one. Using the Directory menu, set the current directory to *diskname*:MPW:Examples:CExamples.

Using the Directory menu, set the directory to *diskname*:`MPW:Examples:CExamples`. Now select Build from the Build menu and type the program name `Memory`. You will see something like the following code on the screen. (A few line breaks and continuation characters have been added for readability.)

```
# 4:12:40 PM ----- Build of Memory.
# 4:12:41 PM ----- Analyzing dependencies.
# 4:12:43 PM ----- Executing build commands.
    C Memory.c

    Link -w -rt DRVW=0 -sg Memory ∂
      "diskname:MPW:Libraries:"DRVRRuntime.o Memory.c.o ∂
      "diskname:MPW:CLibraries:"CRuntime.o ∂
      "diskname:MPW:Libraries:"Interface.o -o Memory.DRVW -c ∂
      "????" -t "????"
    Rez -rd -c DMOV -t DFIL Memory.r -o Memory
# 4:13:06 PM ----- Done.
'Font/DA Mover' 'diskname:System Folder:System' Memory  # Install DA|
```

Press Enter to launch the Font/DA Mover. (If you have two megabytes or less of RAM, you may not be able to launch the Font/DA Mover under MultiFinder: restart with the Command key held down, then try again.) Install the Memory DA in your System File. It gives you the current size of the system heap and the application heap.

The next two sections explain how to build a desk accessory step by step without a Build menu.

Compiling a desk accessory

You compile a desk accessory in exactly the same way that you compile an application or tool. The information given earlier in the chapter regarding include-file search rules, segmentation, and resources applies equally to tools, applications, and desk accessories.

Linking a desk accessory

A desk accessory written in C must be linked with the following libraries:

```
{Libraries}DRVRRuntime.o
{Libraries}CRuntime.o
```

DRVRRuntime.o must be the first object file passed to the linker.

Stand-alone code resources

When you are developing programs for the Macintosh environment, it is often desirable to build stand-alone code resources. For example, you may want to create custom controls. Some of these resources are

`'WDEF'`	window definition procedure (for custom windows)
`'CDEF'`	control definition procedure (for custom controls)
`'LDEF'`	list definition procedure (for List Manager)
`'MDEF'`	menu definition procedure (for custom menus)
`'INIT'`	a code resource that is loaded and run at boot time by the system startup code.
`'XCMD'`	external command for HyperCard™
`'cdev'`	control device (a code resource used by the Control Panel)

Follow these steps to create a stand-alone code resource:

1. No global or static variables can be declared. No calls can be made to library routines that have global variables (such as `printf()`).
2. If you are using string constants in C, you must use the C compiler's `-b` option to put those constants in the code segment (rather than generating global variables).
3. You must use Link's `-rt` option to specify the code resource type (such as `'WDEF'`, `'INIT'`, and so on) and the resource ID.
4. Because most stand-alone code resources are Pascal-style functions, you may need to declare the main procedure with the `pascal` keyword.
5. You must use Link's `-m` option to specify the entry point for the code resource. The procedure that is the main procedure for the stand-alone code resource must be the first procedure in the source file, and that source file's object file must be the first file in Link's list of object files to link. In the case of MPW C, you must make the main entry point the first function in your file (including all `#include` files).
6. If you need to place all of your object modules in one resource (as in the case of a CDEF), use Link's `-sg Main` option to combine several segments into the segment `Main`.

For more information and an example written in both Pascal and C, see the section "Building a Stand-Alone Code Resource" in Chapter 8 of the *MPW Reference*.

Creating code for different models of Macintosh

Using version 3.0 of MPW C, you can create applications that run on all models of the Macintosh. This section outlines the compatibilities among the machines and the strategies for writing and compiling code that will run on the different models. At least two megabytes of RAM are recommended to compile MPW C programs.

Source code

You can write your source code to be compatible with one or more models of the Macintosh. You have four primary options:

- Code written for a Macintosh 512K also runs on a Macintosh XL, a Macintosh Plus, a Macintosh SE, and a Macintosh II. If you want your program to run on any model, follow the recommendations in *Inside Macintosh,* Volumes I through III.
- Code written for a Macintosh Plus also runs on a Macintosh SE and a Macintosh II. If you want your program to run on either of these models, follow the recommendations in *Inside Macintosh,* Volumes I through IV.
- Code written for a Macintosh SE also runs on a Macintosh II and an upgraded Macintosh 512K or Macintosh Plus. If you want your code to run on any model with the most recent system disk, follow the recommendations for a Macintosh SE in *Inside Macintosh,* Volume V.
- Code written for a Macintosh II, using the ROM code that is present only in that model, runs only on a Macintosh II.

Header files

A set of header files provided with MPW C gives you access from C to the User Interface Toolbox and Macintosh Operating System routines built into the Macintosh ROMs. Volume 5 of *Inside Macintosh* describes the ROM code that is new with the Macintosh II and the Macintosh SE. Chapter 4 of this manual describes the C header files.

Much of the new material is usable only on a Macintosh II, because it makes use of hardware options that are not available on other models. You can include all of the header-file definitions in code for use on any model, but you cannot call ROM routines that are not present on the machine that will run the compiled code.

Compiler options

With the addition of the Macintosh II to the product line, there are now compiler differences between the models as well as ROM code differences.

The Macintosh II contains an MC68020 (instead of an MC68000) processor and an MC68881 coprocessor for floating-point operations. By default, the compiler generates code for the MC68000 and uses SANE software routines for floating-point operations: this code will run on any Macintosh.

To take advantage of the more powerful capabilities of the Macintosh II, you can use the `-mc68020`, `-mc68881`, and `-elems68881` command-line options, as explained in the next section.

Creating code for the MC68881 and MC68020

The Macintosh II contains a Motorola MC68881 floating-point coprocessor and an MC68020 processor. These chips offer significant performance advantages over the standard SANE floating-point arithmetic and the MC68000 used by other computers in the Macintosh family.

This section provides technical notes on how the MPW 3.0 C Compiler uses the MC68881 and MC68020. You need this information only if you are writing code that will run on a Macintosh II, and probably only if portions of your code are in assembly language.

For more detailed information, see the *Apple Numerics Manual,* second edition; the *MC68881 Floating-Point Coprocessor User's Manual;* and the *MC68020 32-Bit Microprocessor User's Manual,* second edition.

MC68881 enhancements

The MC68881 floating-point coprocessor executes basic arithmetic and a number of transcendental functions very fast. The Macintosh II always takes advantage of the MC68881 to speed up floating-point arithmetic, regardless of whether the object code it is running was compiled specifically for the Macintosh II. This performance improvement is possible because the compiler issues calls to the SANE arithmetic packages contained in the Macintosh ROMs. The ROMs in the Macintosh II pass some calls on to the MC68881. The ROMs in Macintosh computers that do not contain the MC68881 use their own floating-point routines.

The MPW 3.0 C Compiler offers three options that provide more direct support of the MC68881. When you invoke these options, the compiler generates MC68881 instructions instead of the standard SANE instructions either for basic arithmetic or for both basic arithmetic and transcendental functions.

The MPW 3.0 C Compiler makes direct use of the MC68881 floating-point coprocessor through these command-line options:

- `-mc68881` causes the compiler to generate MC68881 code for basic arithmetic operations, including addition, subtraction, multiplication, division, and comparisons. The results of all functions are the same as would be computed on any model of Macintosh.

 With only the `-mc68881` option invoked, the compiler does not use the MC68881 for transcendental functions or binary-decimal conversions. These functions and conversions are handled by the software SANE library routines described in the *Apple Numerics Manual.*
- `-elems881` causes the compiler to use MC68881 instructions instead of the software SANE routines for transcendental functions. The results are slightly less accurate, but the functions run very fast.

You can invoke the Macintosh II compiler options by including the flags in the compile command line. For example, to invoke the `-mc68881` option and the `-mc68020` option, use this command line:

`c -mc68881 -mc68020` *filename*

From the source text you can tell if the command-line option `-mc68881` has been requested, because the compiler defines `mc68881` when it is requested in the command line.

Linking with the `-mc68881` *option*

Four of the C libraries (CLib881.o, CSANELib881.o, Math881.o, and Complex881.o) are for use only with the MC68881 on the Macintosh II.

If you are using the `-mc68881` compiler option, you must follow these linking rules:

1. Link with {CLibraries}CLib881.o in addition to {CLibraries}StdCLib.o and CRuntime.o. Put CLib881.o before the other libraries in {CLibraries} in the link list, because it contains some definitions that override 80-bit versions in StdCLib.o and CRuntime.o.
2. Link with {CLibraries}CSANELib881.o instead of {CLibraries}CSANELib.o.
3. Link with {CLibraries}Math881.o instead of {CLibraries}Math.o.
4. Link with {CLibraries}Complex881.o instead of {CLibraries}Complex.o.

The `extended` *data format*

Both the software-SANE 80-bit `extended` data format and the Motorola 96-bit `extended` data format meet the IEEE standard for accuracy. The difference is that the 96-bit format contains 16 bits of padding so that data of type `extended` will fit evenly into three 32-bit memory accesses.

Two new type definitions appear in the SANE.h header file: `extended80` and `extended96`. Two new functions are provided to convert between 80-bit and 96-bit formats: `x96tox80` and `x80tox96`.

The 16 bits of padding occupy the space between the exponent field and the significand field. The *Apple Numerics Manual*, second edition, illustrates the two `extended` formats.

MC68881 instructions

With the `-mc68881` option set, the compiler uses MC68881 instructions for addition, subtraction, multiplication, division, conversion between binary formats, and comparison. When the program runs, the MC68881 performs the calculations more quickly than and as accurately as the SANE routines that would otherwise be used.

With the `-elems881` option set, the compiler uses MC68881 instructions for the additional operations `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `exp`, `log`, `fabs`, `log10`, `exp10`, `sqrt`, `atanh`, `exp1`, `exp2`, `log2`, `ln1`, and `trunc`. When the program runs, the MC68881 performs the calculations more quickly but slightly less accurately than the software SANE routines that would otherwise be used.

Floating-point registers

The MC68881 makes available eight additional registers, FP0 through FP7, which store IEEE extended values (that is, `extended` or `long double`. You can use registers FP0 through FP3 for temporary storage in assembly-language routines without restoring their values. If you use registers FP4 through FP7, you must preserve their contents.

With the `-mc68881` option set, the compiler can allocate variables of type `int` and type `extended` (`long double`) to registers. The compiler optimizes performance by using the registers for efficient floating-point evaluation.

MC68020 enhancements

The MPW 3.0 C Compiler provides direct support of the MC68020 processor through the `-mc68020` command-line option. With this option set, the compiler can use the MC68020 instruction set, including integer multiplication and division and bit-field operations.

Chapter 2 The MPW C Language

THE INFORMATION PROVIDED IN THIS CHAPTER SUPPLEMENTS the first edition of Kernighan and Ritchie's *The C Programming Language* (herein called *K&R*, or *K&R 1* to distinguish it from the second edition, called *K&R 2*). The *MPW 3.0 C Reference* describes features of MPW C not described in K&R. Much of this material is derived from the ANSI draft C standard.

This chapter discusses the lexical elements that make up the MPW C language; the operation of the preprocessor; the data types (including `void` and enumerated types), specifiers like `register` and qualifiers like `const`, `volatile`, and `pascal`; conversions and evaluation order; and functions (including Pascal-style functions). It also has a section on implementation details for those writing performance- or hardware-oriented code. It also discusses the MPW C interface to the Standard Apple Numerics Environment (SANE). ■

Contents

Language definition 33
Lexical conventions 34
- Keywords 34
- Character constants: newline, carriage return, and vertical tab 34
- Ellipsis character 35
- Identifiers 35
- Numeric constants 35
- String constants 36
 - Pascal-style strings 36
 - String concatenation 37
- Line-end comments (Apple extension) 37

The preprocessor 37
- The `#include` directive 38
- Reserved symbols 38
- Conditional compilation 39
- Predefined symbols 39
- Pragmas 40
 - Segmentation 40

Dumping and loading precompiled information 41
Unused variables and parameters 41
Stringization 42
Tokenization 42
Types 43
Scalar data types 43
Integral types 44
Pointer types 45
SANE types 45
Type `void` 45
Enumerated types 46
Structures 46
Bit-fields 47
Type qualifiers 47
`const` 47
`volatile` 48
`pascal` 50
The `register` class specifier 50
Integer conversions and arithmetic operations 50
Sequence points 51
Evaluation order 52
Function declarations and definitions 53
Pascal-style functions 55
Calling Pascal-style functions 55
Writing Pascal-style functions 56
Declaring and using Pascal-style function pointers 57
Direct functions 57
Parameters to the `main` function 59
Compiler implementation details 59
Size and byte alignment of variables 60
Byte ordering 60
Variable allocation and register use 61
Bit-fields 61
Bitwise operations 62
Compiler limitations 62
Structure results 63
Multiple external definitions 63
Standard Apple Numerics Environment extensions 63
Expressions 66
Comparisons involving a NaN 66

Functions 66
Numeric input/output 67
Numerics environment 67
About the C SANE Library 67
Programming with IEEE arithmetic 68

Language definition

This section describes the MPW 3.0 C language, including language features such as `void` and enumerated types, the SANE data types, function prototypes, and Pascal-style functions.

C was originally defined in *K&R 1:* this original dialect of C is commonly called "K&R C."

Since then, a number of extensions have been added to the language. The more durable of these extensions have been incorporated into the ANSI draft C standard. In addition, the standard contains some entirely new features, such as function prototyping, which strengthens C's type-checking.

MPW 3.0 C contains many of the ANSI C language features: these features are described later in this chapter.

The ANSI draft C standard leaves numerous details of the language definition up to the implementation. (These details are labeled *implementation-defined.*) Other sections of this chapter fill in many of these details.

- Where the standard's language definition leaves choices to the implementers, this chapter describes how these aspects of C have been implemented on the Macintosh.
- Where Apple Computer, Inc., has modified or extended the standard's language definition, this chapter documents the changes.

In addition to the ANSI features, MPW 3.0 C contains a number of Apple extensions, notably SANE. These extensions are described later in this chapter.

Lexical conventions

This section describes the lexical elements of MPW C. These include constants, identifiers, and strings.

Keywords

The identifiers listed below are reserved for use as keywords, and may not be used otherwise. ANSI C keywords are in **bold**. Apple extensions are underlined.

auto	**do**	**for**	**short**	**unsigned**
break	**double**	**goto**	**signed**	**void**
case	**else**	**if**	**sizeof**	**volatile**
char	**enum**	**int**	**static**	**while**
comp	entry‡	**long**	**struct**	
const	extended	pascal	**switch**	
continue	**extern**	**register**	**typedef**	
default	**float**	**return**	**union**	

‡ Reserved for future use.

Character constants: newline, carriage return, and vertical tab

In MPW windows or TextEdit, the **newline** character advances the cursor to the left margin on the next line. In C notation, newline is represented by `\n`. The character code for `\n` in MPW 3.0 C is the ASCII value (13) for the carriage-return character. In C notation, the **carriage-return** character is represented by `\r`. The character code for `\r` in MPW 3.0 C is the standard ASCII value (10) for the line-feed character. (These assignments are reversed from most other implementations of C.)

In MPW 2.0 C, the escape characters `\n` and `\r` both represented the carriage-return character (ASCII value 13). This often caused duplicate `switch` labels, and conflicts with the ANSI draft C standard. In MPW 3.0 C, `\n` and `\r` have unique values.

The vertical-tab character (ASCII 31), represented by `\v`, is not meaningfully interpreted in MPW windows or by TextEdit.

The following characters are interpreted by the compiler as space characters: Option-space (ASCII 202); line-feed (ASCII 10); and backspace (ASCII 8). Other control characters (below ASCII 31) are not accepted. Characters from the Macintosh extended character set are accepted except as noted above.

There is no mapping of characters in character constants and string literals. All characters and escape sequences can be represented.

Ellipsis character

The **ellipsis character** used in function prototypes (`...`) is a series of three periods. The Macintosh ellipsis character (..., produced by typing Option-;) may not be used instead.

Identifiers

MPW C identifiers are limited to 63 characters. Both the MPW 3.0 C Compiler and the MPW 3.0 Linker impose the same limit. Identifiers are case sensitive.

Numeric constants

Numeric constants that include floating-point syntax—a decimal point (.), an exponent field, or a following `F`—or that lie outside the range of type unsigned `long` are of type `extended`. Decimal-to-binary conversion for numeric constants is done at compile time and hence is governed by the default numerics environment (see "Numerics Environment" later in this chapter).

A numeric constant is treated according to its value:

- An integer constant within the range of `long` (see Table 2-1) is treated as a `long`.
- An integer constant outside the range of `long` but inside the range of `unsigned long` is treated as an `unsigned long`.
- A numeric constant that contains a decimal point or is written in exponential notation is always treated as type `extended`.
- The type specifier `l` (or `L`) is redundant but supported.
- The type specifier `u` (or `U`) causes a integer constant to be treated as `unsigned long`.
- The type specifier `f` (or `F`) causes a floating-point constant to be treated as `extended`.

Overflows of integer constants and variables are not detected. For example, consider these initialization statements:

- `long i = 4000000000;`

 Because 4,000,000,000 is too big to be stored as a signed 32-bit value, the compiler cannot store it in the signed `long` variable `i`. Although this statement does not result in an error, its results should be considered unpredictable. Because 4,000,000,000 lies inside the range of `unsigned long`, the compiler assigns to `i` the sequence that represents the unsigned integer 4,000,000,000 (0xEE6B2800). The value of `i` is then the negative signed integer (–294,967,296) represented by that bit string.

- `unsigned long i = 4000000000;`

 Because 4,000,000,000 fits into a 32-bit unsigned field, this initialization statement is correct.

- `unsigned long i = 6666666666;`

 `long i = 6666666666;`

 Because 6,666,666,666 is outside the ranges of both `long` and `unsigned long`, the compiler stores it as 0x8D5D42AA, which is the correct value with the leading 1 missing.

◆ *Note:* Type `short int` is equivalent to `short`, and type `long int` is equivalent to `long`.

String constants

String constants and strings created by string concatenation are limited to 512 characters.

Pascal-style strings

The escape sequence `\p` generates a length byte when it appears at the beginning of a string literal. Strings beginning with `\p` have both a length byte at the front and a null terminator at the end. The length does not include either the length byte or the null terminator, just the characters in between.

For example, the definition

```
unsigned char * s = "\pabc";
```

generates a string that begins with a length byte of 3, and is terminated by a null byte. String `s` can be used directly as a Pascal string, and `&s[1]` or `(s+1)` can be used as a C string.

String concatenation

When the preprocessor encounters two successive string constants, it concatenates them into one. This feature can be handy for splitting a lengthy string constant across several source lines:

```
printf("Let us go then, you and I\n"
       "When the evening is spread out against the sky,\n"
       "Like a patient etherised upon a table\n");
```

String concatenation allows you to format your source the way you want it to look. No continuation characters are needed, and no extra spaces will be inserted.

◆ *Note:* The maximum length of a concatenated string is 512 characters.

Line-end comments (Apple extension)

In addition to the traditional comments `/* comment */`, MPW 3.0 C also allows comments preceded by // at the end of a line, as in the following examples:

```
//This is a comment. It ends at the end of this line.
//Each line of the comment needs the // delimiter.

void eatchar(void) // This function eats characters.
{
   int c;
   c = getchar();
}
```

Anything between the // and the end of the line is ignored.

◆ *Note:* This feature is an Apple extension. It is not specified by the ANSI standard.

The preprocessor

Before a program is compiled, the preprocessor reads the source files, expands macros, executes `#include`, `#define`, and `#pragma` directives, and performs other operations, as described in the sections that follow.

The `#include` directive

MPW 3.0 C allows `#include` directives to be nested 15 levels deep.

Reserved symbols

MPW 3.0 C supports the following reserved symbols:

`__FILE__` is a reserved preprocessor symbol whose value is a character string consisting of the current filename.

`__LINE__` is a reserved preprocessor symbol whose value is the current line number within the current source file.

`__DATE__` is a reserved preprocessor symbol whose value is the current date.

`__TIME__` is a reserved preprocessor symbol whose value is the current time.

`__DUMP__` is a reserved preprocessor symbol whose value is a character string consisting of the current filename with its suffix replaced with the string `".DUMP"`. It is used in conjunction with the `load` and `dump` pragmas.

`__SEG__` is a reserved preprocessor symbol that overrides the segment name associated with functions that follow; it remains in effect until the next `__SEG__` directive. The default segment name, `Main`, may also be overridden with the `-s` compiler option. (This form of segmentation is not recommended: see the description of the `segment` pragma, below.)

`__STDC__` is a reserved preprocessor symbol whose value is 0.

The symbols `__FILE__`, `__LINE__`, `__DATE__`, `__TIME__`, `__DUMP__`, and `__SEG__` all begin and end with two underscore characters.

Conditional compilation

In addition to the usual conditional compilation directives, MPW 3.0 C provides two additional ones, as specified by the ANSI draft C standard.

`#elif` The preprocessor equivalent of `else if`.

`defined` The expression `defined(`*name*`)`, in a `#if` or `#elif` statement, has the value 1 if *name* is defined; 0 otherwise. For example,

```
#if defined(MAKEDUMP)
```

is 1 if `MAKEDUMP` is defined, 0 otherwise. This statement is equivalent to

```
#ifdef MAKEDUMP
```

Up to 40 conditional compilation directives may be nested in MPW 3.0 C.

Predefined symbols

The following symbols are predefined for use in conditional compilation:

```
MC68000
mc68000
m68k
macintosh
applec
mc68881
```

The `mc68881` symbol is defined only when the `-mc68881` command-line option is used.

Each of the predefined symbols has the value 1, as if a statement of this form had appeared at the beginning of the source code:

```
#define MC68000 1
```

You can undefine any of the predefined symbols with the `-u` compiler option.

Pragmas

Pragmas are a feature introduced by the ANSI draft C standard to allow implementors to define instructions to specific compilers without creating portability problems. Preprocessing directives of the form

`#pragma` *tokens*

are implementation-specific instructions to the compiler. They control various aspects of compilation (such as the target processor, optimizations, debugging information, and listings), that are outside the scope of the C language definition.

MPW 3.0 C supports a number of pragmas, which control various aspects of compilation. Except for `unused`, MPW C pragmas are expected between external definitions, and affect the definitions that appear following them in the source. Because pragmas are preprocessing directives, they are on separate source lines and may appear between any two tokens in the source. Unrecognized pragmas are ignored.

Segmentation

The `segment` pragma controls the program's segmentation. Segment names are case sensitive. The default segment name is `Main`. The most recent segmentation pragma has effect.

`# pragma segment` *segmentname*

places the generated code in segment *segmentname*. The linker produces a separate `'CODE'` resource for each distinct segment. (There should be only one space before *segmentname:* if there are more spaces, the extra spaces will be taken as part of the name.)

This command is equivalent to the deprecated older form

`#define _ _SEG_ _` *segmentname*

and is preferred over it: the older form is an anachronism, and may not be supported in future compilers. The default segment name, Main, may also be overridden with the `-s` compiler option. (As above, there should be only one space before *segmentname:* if there are more spaces, the extra spaces will be taken as part of the name.)

Dumping and loading precompiled information

The `dump` and `load` pragmas are useful for creating precompiled versions of header files that allow header information to be processed more quickly. When precompiled information is loaded, the values of the options `-sym`, `-n`, and `-mc68881` must be the same as they were when the precompiled information was originally dumped. (These options are described in Appendix A.)

`#pragma dump "`*filename*`"`	dumps all global symbols (function names; variables, struct, union, and enum tags; and macro definitions) into *filename* in a compressed format.
`#pragma load "`*filename*`"`	reads the file created by the `#pragma dump` *filename* command.

Here is an example:

```
#if !defined(USEDUMP)
   #include <types.h>
   #include <Quickdraw.h>

   #if defined(MAKEDUMP)
      #pragma dump "myDumpFile" /* create dumpfile for subsequent
                                          compilations */
   #endif
#else
   #pragma load "myDumpFile"
#endif

main()
{
   ...
}
```

Unused variables and parameters

The `unused` pragma suppresses compile-time warnings that are emitted when the compiler discovers that one or more local variables or parameters have not been used within the body of a function definition. This pragma can only be within a function body.

`#pragma unused (`*var_or_param* [, *var_or_param*]...`)`

For example, the code fragment

```
void foo (int *a, int *b)
{
   *a = 1;
}
```

produces the compiler warning

```
Warning 238: Parameter "b" not used within the body of function:foo
```

You can suppress this warning with the modified fragment

```
void foo (int *a, int *b)
{
#pragma unused(b)
   *a = 1;
}
```

The `unused` pragma is a safer way of suppressing this compiler warning than using the `-w` option, which suppresses all warnings.

Stringization

Stringization is an ANSI feature of the preprocessor. If a macro definition contains `#` *identifier* in its replacement list, `#` *identifier* is replaced by *identifier* changed into a string literal (hence the name *stringization*). Consider this code fragment, which defines and uses `PRINTVAR`, which prints the name and value of a variable:

```
#define PRINTVAR(v) printf(#v " = %d\n", v)
PRINTVAR(var);
```

After stringization, this fragment of code becomes

```
printf("var" " = %d\n", var);
```

After string literals are catenated, it becomes

```
printf("var = &d\n", var);
```

Stringization is defined in section 3.8.3.2 of the ANSI draft C standard (May 11, 1988).

Tokenization

Tokenization is an ANSI feature of the preprocessor. If a macro definition contains *token* `##` *token* in its replacement list, the tokens are concatenated lexically, forming one token. Consider this fragment:

```
#define xn(x, n) x ## n
   xn(a, 3) = 5;
```

After tokenization, this code becomes

```
a3 = 5;
```

Tokenization is defined in section 3.8.3.3 of the ANSI draft C standard (May 11, 1988).

Types

MPW C's data types include scalar data types, pointer types, type `void`, enumerated types, and structures.

Scalar data types

Each of the scalar data types is defined in Table 2-1. On the MC680x0, the least significant byte of the integer data types has the highest memory address.

■ **Table 2-1** Size and range of data types

Data type	Bits	Minimum	Maximum	Description
`char`	8	–128	127	Same as `signed char`.
`unsigned char`	8	0	255	Macintosh character set.
`short`	16	–32,768	32,767	Same as `signed short`, `short int`, and `signed short int`.
`unsigned short`	16	0	65,535	
`int`	32	–2,147,483,648	2,147,483,647	Same as `signed int`. In MPW C, `int` and `long` are the same size.
`unsigned int`	32	0	4,294,967,295	
`long`	32	–2,147,483,648	2,147,483,647	Same as `signed long`, `long int`, and `signed long int`. In MPW C, `long` and `int` are the same size.
`unsigned long`	32	0	4,294,967,295	
`enum`	8	–128	127	Size depends on range of literals.
`enum`	16	–32,768	32,767	Size depends on range of literals.
`enum`	32	–2,147,483,648	2,147,483,647	Size depends on range of literals.
Ctype `*`	32	0	4,294,967,295	Pointer to *Ctype* .
`float`	32	1.5E–45	3.4E+38	IEEE single-precision floating point (preferred).
`single`				IEEE synonym for `float`.
`double`	64	5.0E–324	1.7E+308	IEEE double-precision floating point.
`extended`	80	1.9E–4951	1.1E+4932	IEEE extended-precision floating point (default: 80 bits allocated when `-mc68881` *not* active).

(continued)

■ **Table 2-1** (continued) Size and range of data types

Data type	Bits	Minimum	Maximum	Description
`extended`	96	1.9E–4951	1.1E+4932	IEEE extended-precision floating point (96 bits allocated when `-mc68881` active).
`long double`				ANSI synonym for `extended`.
`comp`	64	≈–9.2E18	≈9.2E18	MPW C SANE signed integer.

Integral types

MPW C supports these integral types (in addition to the enumerated types):

`char` (or `signed char`)	`unsigned char`
`short` (or `signed short`)	`unsigned short`
`long` (or `signed long`)	`unsigned long`
`int` (or `signed int`)	`unsigned int`

Each of the integral types is signed by default: the keyword `signed` is optional in declaring the signed types. The keyword `unsigned` is required in declaring the unsigned types.

Both variables and literals of type `char` are signed. (This is a change from MPW 2.0 C: see Appendix D for details.) The type `unsigned char` is more useful when representing the Macintosh character set.

Type `wchar_t`, the ANSI "wide character" type designed to support 16-bit characters, is `unsigned short`.

```
typedef wchar_t unsigned short;
```

◆ *Note:* The Script Manager does not use the `wchar_t` type to implement 16-bit characters. The Script Manager must accommodate strings containing both 8-bit and 16-bit characters, so it uses a different strategy, which is described in the Script Manager chapter of *Inside Macintosh,* Volume V.

In MPW C, the `int` and `long` types are the same size.

In MPW C, the type of the `sizeof` operator is `int`.

Pointer types

The size of pointers is 32 bits, as are types `int` and `long`. Thus, when pointers are cast to types `int` or `long`, and vice versa, the bit pattern remains unchanged.

MPW 3.0 C considers `char *` and `unsigned char *` to be type compatible: for instance, you can pass a variable of type `char *` to a function expecting `unsigned char *` without first casting the variable to `unsigned char *`.

Type `size_t`, the type that is returned by `sizeof` and is required to hold the maximum size of an array, is of type `unsigned int`. Type `ptrdiff_t`, the type that is required to hold the difference of two pointers to members of the same array, is of type `int`.

```
typedef unsigned int size_t;
typedef int ptrdiff_t;
```

SANE types

The floating point types and type `comp` are described in more detail in the *Apple Numerics Manual*. Type `extended` is a synonym for `long double`.

Type `void`

The **`void`** keyword tells the compiler that the function being declared does not return a value. Calls to functions of type `void` may not be used in expressions, where a value is required. (See "Pascal-style Functions" later in this chapter.) Type `void` can be used in these cases:

- To define or declare a function that returns no value:

  ```
  void exit(int status);
  ```

- To define or declare a function that takes no parameters:

  ```
  int getchar(void);
  ```

- In a cast, to explicitly discard the return value of a function call:

  ```
  (void) printf("Hello");
  ```

- To create generic pointers. A pointer to `void`, denoted `void *`, can be regarded as a pointer to anything. Such a generic pointer can result, for example, from a call to `malloc`. In K&R C, `malloc` returned `char *`, but there was no particular reason to single out `char` for this honor, as `malloc` had no way of knowing what it was allocating space for. The ANSI standard provides a cleaner solution: `malloc` returns a result of type `void *`:

  ```
  void *malloc(size_t size);
  void *realloc(void *p, size_t size);
  ```

- It is illegal to dereference pointers to `void` variables.

Enumerated types

A C enumeration, analogous to a Pascal enumerated type, is a unique type marked by the type qualifier `enum` whose values range over a set of named integral constants. The declaration

```
enum sea {Mediterranean, Adriatic, Aegean=10, winedark};
```

makes `sea` an enumeration tag for the set of values { 0, 1, 10, 11}.

Each of the enumeration constants is defined by an integral constant expression whose value can be represented by an `int`.

(Enumerated types are discussed in considerable detail in the ANSI draft C standard and in *K&R 2.*)

Structures

Structures may be assigned, passed as parameters, and returned as function results, as described in the ANSI draft standard and in *K&R 2.* The left and right sides of a structure assignment must naturally have the same type.

One of the implementation-defined features of an ANSI C compiler is what happens when you access a member of a union by using a member of a different type. In MPW 3.0 C, the data thus accessed will be reinterpreted as if it were of the accessing type. For example, consider this union and a few statements using it:

```
union foo {
   int i;
   struct {
      short a,b;
   } sh;
} example;

example.i = 0x0010FFFF;
printf("a = %d, b = %d\n", example.sh.a, example.sh.b);
```

This example prints `16` and `-1` as expected.

Bit-fields

Bit-fields in structures and unions may have type `int`, `signed int`, or `unsigned int`. Bit-fields of type `int` are signed by default. (This default is incompatible with MPW 2.0 C: see the section "Bit-fields," in Appendix D.)

You declare a signed bit-field in one of these two ways. You can use the second method because the default is signed:

```
struct {signed int field:2;};
struct {int field:2;};
```

You declare an unsigned bit-field like this:

```
struct {unsigned int field:2;};
```

The `&` operator cannot be applied to a bit-field in a structure.

Since bit-fields are based on type `int`, their length is limited to 32 bits.

Type qualifiers

ANSI C adds a new semantic category: type qualifiers. A type qualifier can modify any of the standard types, and give it specific characteristics, such as permanence or impermanence.

`const`

The `const` type qualifier declares a constant object of a given type:

```
const int x = 3;

x = 7;    // illegal!
x += 1;   // illegal!
```

The `const` qualifier can be used to declare a pointer to a constant, as here:

```
const int *ptr_to_constant;
int x = 3;
ptr_to_const = &x;
*ptr_to_const += 1;  // illegal!
```

Here, `*ptr_to_constant` is a pointer to an object that cannot be changed by assignment; however, the pointer itself can be changed to point to another `const int`. Some examples would be a constant in ROM or a read-only location in low memory.

The `const` qualifier can also be used to declare a constant pointer, as here:

```
char *const const_ptr = "Hello,";
const_ptr = " world";                // illegal!
```

Here `const_ptr` is a fixed pointer to a location. The value of the pointer cannot be changed, but the value of the object it points to can be changed.

In fact, you can make a constant pointer to a constant value, as here:

```
const char *const constant_ptr = "Hello, world";
constant_ptr   = "Goodbye, Mr. Chips";          // illegal!
constant_ptr[0] = 'X';                          // illegal!
```

Here `constant_ptr` is a constant pointer to a constant location: neither the pointer nor the value of the object it points to can change.

volatile

The `volatile` type qualifier is used to suppress certain optimizations: notably storing a commonly used value in a register for future use. The declaration

```
volatile int x;
```

tells the compiler not to store the value of `x` in a register. The `volatile` qualifier is typically used in conjunction with the `setjmp` and `longjmp` functions or with hardware-dependent memory locations that may change without warning. Without the qualifier, the compiler might place a variable in a register that might be overwritten by the `longjmp` function.

The following example shows the use of the `volatile` qualifier:

```
#include <types.h>
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

/*
** Typical usage of "volatile" qualifier with setjmp/longjmp
**
*/
void *new_node(unsigned long size)
{
   volatile void *node = nil;
   int j;
   jmp_buf jbf;

   /*
   ** Failure exit, free memory if allocated
   **
   ** node MUST be declared "volatile", if it
   ** is in a register, longjmp will trash it
   */
   if (j = setjmp(jbf)) {
      if (j==1)
         fprintf(stderr, "new_node fails: insufficient memory");
      else if (j==2)
         fprintf(stderr, "new_node fails: add_node fails");
      else
         fprintf(stderr, "new_node fails: unspecified error");

      if (node != nil) free(node);
      return nil;
   }

   /*
   ** Allocate node
   */
   node = malloc(size);
   if (node == nil) longjmp(jbf, 1);

   /*
   ** Attempt to add node to some sort of list
   */
   if (add_node(node, size) < 0) longjmp(jbf, 2);

   return node;
}
```

- *Note:* The `volatile` specifier in MPW 3.0 C does not behave as specified in the ANSI draft C standard. Future MPW C compilers may behave differently from MPW 3.0 C.

`pascal`

The `pascal` type qualifier declares that a function is a Pascal-style function and follows the calling conventions for such functions. Pascal-style functions are described in a section of their own, later in this chapter. Pascal-style calling conventions and the correspondence between Pascal and C types are defined in Appendix C.

The `register` class specifier

The `register` class specifier tells the compiler to store a variable in a register if possible. The use of this specifier is more restricted than in K&R C. If an object is declared `register`, you can't apply the unary `&` (address of) operator to it. This is true whether or not the compiler actually puts this object in a register.

Integer conversions and arithmetic operations

Conversions between integral types can occur either implicitly, as by combining different types in an expression, or explicitly, as by a cast. In some conversions, one must choose whether to preserve a number's value, or its unsignedness. MPW 3.0 C follows the ANSI draft C standard in adhering to the principle of value-preserving, rather than unsigned-preserving, when the two conflict.

In any expression where you can use an `int` or `unsigned int`, you can use a `char`, a `short`, or an `int` bit-field, their signed or unsigned varieties, or an object with enumeration type. If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise it is converted to an `unsigned int`.

When an unsigned integer is converted to another integral type, either implicitly or by a cast, and if the value can be represented by the new type, its value is unchanged.

When an unsigned integer is converted to a signed integer of equal size, and the value cannot be represented, the bits are unchanged: they are simply reinterpreted as a signed integer.

When a signed integer is converted to an unsigned integer with equal or greater size, if the value of the signed integer is nonnegative, its value is unchanged. If the value is negative and the unsigned integer has greater size, the signed integer is first promoted to the signed integer corresponding to the unsigned number, then sign-extended.

When a value from an integer type is demoted to an unsigned integer type with smaller size, the high-order bits are truncated. When an integer is demoted to a shorter signed integer or an unsigned integer is converted to its corresponding signed integer; if the result cannot be represented, the high-order bits are truncated.

The remainder on integer division has the same sign as the dividend.

The result of a right-shift of a negative signed integer is also negative.

When an integer is demoted to a shorter signed integer, the least-significant bits are retained.

Sequence points

A **sequence point** is a concept used by the compiler to determine evaluation order and for optimization purposes. Between two sequence points, an object's stored value can be modified no more than once. Sequence points are located

- at a function call, after the arguments have been evaluated
- after the first operand of the logical AND (`&&`), logical OR (`||`), conditional (`?`), or comma (`,`) operators
- at the end of full expressions

A **full expression** is one of the following:

- an initializer
- an expression statement, like an assignment
- the controlling expression of an `if`, `switch`, `while`, or `do` statement
- each of the three expressions of a `for` statement
- the expression in a `return` statement

As a result, the result of a statement like

```
i = ++i + 1;
```

is undefined, whereas the result of a statement like

```
i = i + 1;
```

is well defined.

Evaluation order

MPW C does not alter the order of evaluating associative operations. In this respect, MPW C (and ANSI C) differs from K&R C. The significance of this change is most apparent in floating-point arithmetic. Take the floating-point expressions `(a + b) + c` and `a + (b + c)`. They should be equal for all values of `a`, `b`, and `c`. However, floating-point arithmetic is not really real arithmetic. Assume that `b` is greater than 0, `a` equals `-b`, and `c` is positive but substantially smaller than `b`. (Specifically, assume that `c/b` is smaller than the smallest number that can be represented in the type used.) Now, `(a + b) + c` is `0 + c`, or `c`, whereas `a + (b + c)` equals `a + b`, or `0`. In other words, floating-point addition is not associative. Similar demonstrations hold for multiplication, and for the commutative and distributive properties.

Because floating-point numbers do not always have the commutative, associative, and distributive properties, MPW C always honors the evaluation order specified by parentheses in the source text. This general principle, called **"honor thy parentheses,"** is required by the Standard Apple Numeric Environment; it is not required by the ANSI draft C standard, but does not conflict with it. This principle can be used to ensure that compiler optimizations don't disturb algorithms that are carefully designed for maximum accuracy.

MPW C does not define the evaluation order of expressions with side effects, such as function calls and the `++` and `--` operators. Specifically, when evaluating an expression modifies the value of variable as a side effect, and when the variable is also used at another point in the same expression, the value used may be either the value before, or the value after, modification. Programs should not rely on the order of evaluation in these situations. The function call

```
f(i, i++)
```

is an example of an expression whose value is undefined.

Function declarations and definitions

One of the most important additions made by the ANSI draft C standard is in the area of function declarations. By declaring the types as well as the names of all parameters, the new style of declaration (called a function prototype) facilitates type-checking by the compiler. A new-style function declaration has the form

function-name (*parameter-type list*)

The syntax of the parameters is

parameter-type-list:
 parameter-list
 parameter-list , . . .

parameter-list:
 parameter-declaration
 parameter-list , *parameter-declaration*

(For details, see the ANSI draft C standard or K&R 2.)

Function prototypes allow MPW C to check function arguments for consistency and to convert the actual arguments to match the declared types of the formal parameters. We strongly recommend that each use of a function be preceded by a function prototype or a function definition. The nonprototype form is still allowed, to avoid breaking existing code. You are strongly urged, however, to use prototypes in all code, because future versions of MPW C will be less tolerant of old-style function declarations: you may get a warning now, but an error later. (The `-r` compiler option produces a warning for each missing prototype.)

Here is a typical function prototype:

```
long double hypot(long double x, long double y);
```

This prototype declares a function that takes two `long double` arguments and returns a `long double` result. The `x` and `y` are optional: human readers will likely find them useful, but the compiler ignores them.

When prototypes are used:

- Functions must be declared with a prototype before they can be called. Multiple prototypes must agree exactly.
- Actual arguments must agree or be convertible to formal arguments, or a syntax error occurs. Functions that take a variable number of arguments must be declared explicitly, by use of the ellipsis (, . . .).

A new-style function definition is just like a prototype with a body added. A new-style function definition for the `hypot` function might be

```
long double hypot(long double x, long double y)
{
   return sqrt(x*x + y*y);
}
```

MPW C also accepts old-style function definitions: for the `hypot` function we might have

```
long double hypot(x,y)
   long double x,y;
{
   return sqrt(x*x + y*y);
}
```

Here the new-style and old-style *definitions* are equivalent.

MPW C also accepts the old-style declaration syntax without formal parameter type specifications:

```
long double hypot();
```

The new-style and old-style *declarations* are not completely equivalent, even though they serve the same purpose. When old-style declaration syntax is used:

- Functions are declared implicitly by appearing in a call.
- Actual arguments are converted before the call.
- Arguments are not checked for type agreement. The compiler assumes any function can take an arbitrary number of arguments, each of arbitrary type.

If both types of declaration are used to declare the same identifier, the prototype declaration and the nonprototype declaration need not agree except for the result type and number of parameters.

All callable functions are now declared within the CIncludes files. They use function prototypes to describe all parameters. These parameters are now checked for correct type by the C compiler.

Pascal-style functions

A function or procedure written in Pascal (or written in C or assembly language, following Pascal calling conventions) can be called from C. The function-calling conventions used by MPW C and Pascal differ in the order of parameters on the stack, the type coercions applied to parameters, and the location of the return result. C has been extended to allow function calls between these languages.

The `pascal` type-specifier indicates a **Pascal-style function:** `pascal` may be used in declarations and definitions of functions and in pointers to functions; it indicates that the function uses Pascal calling conventions. (Both C and Pascal calling conventions are described in Appendix C.)

The `pascal` keyword can be used in three distinct ways:

- external declarations of functions written in Pascal, to be called from C
- definitions of functions written in C to be called from Pascal (or C)
- pointers to functions that use Pascal calling conventions

These declarations may appear in any scope. Of course, external `pascal` functions may actually be written in assembly language, provided they follow Pascal calling conventions. Similarly, `pascal` functions written in C can be called from assembly language by using Pascal calling conventions.

Pascal-style function declarations are used in the *Inside Macintosh* Libraries to allow C programs to directly call Macintosh library routines that use Pascal calling conventions. Some of these routines are direct functions (described under that name, below).

The following sections describe typical uses of `pascal` declarations and definitions.

Calling Pascal-style functions

You will often want to call a function that uses Pascal-style calling conventions: toolbox functions, for example, or functions written in Pascal. Such functions need to be declared so that your program can call them, using a Pascal-style function declaration.

Here is an example of such a function, `DoCreate`, as defined in Pascal:

```
PROCEDURE DoCreate(fileName: Str255; vRefNum: INTEGER;
                   creator: OSType; fileType: OSType;
                   VAR result: OSErr);
```

Here is the corresponding C function declaration for the `DoCreate` procedure:

```
pascal void DoCreate(const Str255 fileName, short vRefNum,
                     OSType creator, OSType fileType,
                     OSErr *result);
```

Note that the declaration contains the `void` keyword because in Pascal a procedure does not return a value. For a Pascal function, `void` should be replaced with the appropriate type specifer, as shown in Table C-1.

Writing Pascal-style functions

You will often want to write a function that uses Pascal-style calling conventions, so that it can be called by the toolbox or by functions written in Pascal. Such functions need to be defined, using a Pascal-style function definition.

A C function definition (the actual function), like a function declaration, can also be preceded by the `pascal` specifier. The function is then given Pascal-style calling conventions by the compiler. For example, the following C function can be called from Pascal:

```
pascal void MyText(short byteCount, Ptr textAddr,
                   Point numer, Point denom)
   {
      ...
   }
```

The corresponding Pascal function declaration is

```
PROCEDURE MyText(bytecount: integer; textAddr: Ptr;
                 numer, denom: Point);
```

For compatibility with Pascal and assembly language, the compiler converts the names of Pascal-style functions to uppercase before writing them to the object file. When they are called in C programs, these routines should be capitalized exactly as they were declared in C. Pascal-style functions whose names differ only in their capitalization will become duplicate declarations when their names are converted to uppercase by the compiler; therefore such names should be avoided.

All calls documented in *Inside Macintosh* are now declared to be of type `pascal`. This convention allows parameters to be passed directly to the ROM routines in the manner they expect. It also allows the use of a single library, Interface.o, to contain the interfaces to the Macintosh operating system and toolbox.

Declaring and using Pascal-style function pointers

Often you need to call code in a stand-alone code resource, separately compiled from your code: for example, in a window definition (`'WDEF'`) or a package (`'PACK'`). Usually you would read the resource into memory, lock it down, then take its location in memory or a fixed offset from that location and use the resulting address as your function pointer. Simply follow these steps:

1. Declare a pointer to a Pascal-style function from C:

```
pascal void (*drawCharProc) (short c);
                        /* pointer to Pascal function */
```

2. Initialize the function pointer:

```
drawCharPRoc = ...
```

3. Call the function:

```
(*drawCharProc) ('A');
```

Direct functions

In calls to direct functions, one or more 680x0 instructions replace the usual subroutine call (JSR) instruction in the calling sequence. Direct functions are used to specify traps to the Macintosh ROM. Multiple 680x0 instructions may be used to specify package traps. Direct functions can also be used to access specialized machine instructions using in-line code. Direct functions may have either the normal or Pascal calling conventions.

In a direct function definition, the function body is replaced by a direct function initializer. The syntax is similar to the initialization of variables:

direct-function-initializer:
 = *instruction*
 = { *instruction-list* }

instruction-list:
 instruction
 instruction-list, instruction

Each *instruction* must be in the range of either type `short` or type `unsigned short` (that is, –32,768 to 65,535), and is interpreted as a 16-bit 680x0 instruction. These instructions replace the JSR instruction in the calling sequence for the function. For example, the two definitions

```
pascal void PenSize (short width, short height) = 0xA89B;
pascal TEHandle TEStylNew(Rect *destRect, Rect *viewRect) = 0xA83E;
```

define Pascal-style functions that are direct traps to the Macintosh ROM.

A list of instructions, in braces, is also permitted. These two examples are multiple direct functions

```
pascal long SetCurrentA5(void) = {0x2e8d,0x2a78,0x0904};
pascal void SetStylHandle(TEStyleHandle theHandle, TEHandle hTE)
                          = {0x3F3C,0x0005,0xA83D};
```

The unary & (address-of) operator may not be applied to direct functions. Thus it is impossible to create pointers to direct functions or to pass direct functions as parameters.

◆ *Note:* Direct functions can in effect be used for limited in-line machine-language programming, although this practice is not recommended for the faint of heart. When the C compiler encounters a direct function call, it is treated exactly as a function call, that is, the compiler saves any values needed from the "scratch registers," pushes parameters and space for the result if necessary, and then inserts, instead of a JSR, the instruction list from the direct function call. Finally, the scratch registers are restored to their previous values. (Thus a direct function can use the scratch registers without restoring them.)

Parameters to the `main` function

The function `main` may be defined with no parameters, with two parameters, or with an optional third parameter. The names of the parameters are arbitrary. Here are the three kinds of declarations:

```
int main (void);
int main (int argc, char *argv[]);
int main (int argc, char *argv[], char *envp[]);
```

In applications, `argc` is always 1, `argv[0]` contains a pointer to a string containing the filename of the application, `argv[1]` is NULL, and `envp[0]` is also NULL.

In MPW tools, `argc` is the number of arguments, which is always at least 1. Parameter `argv` is an array of pointers to C strings that represent words from the MPW Shell's command line, after aliasing, variable substitution, filename expansion, and the like have been performed. Element `argv[0]` points to the command name. Element `argv[argc]` is always NULL. The third parameter, `envp`, is the environment pointer, a null-terminated array of pointers to `char`. Each element of the array points to a name-value pair, two concatenated strings of the form

name\0*value*\0

Each pair represents the name and value of an exported MPW Shell variable. The C Library function `getenv` is a more portable way to obtain the value of shell variables.

Parameters `argc`, `argv`, and `envp` and the strings pointed to by `argv` and `envp` may all be modified. Any attempt to increase the size of `argv` or `envp` or the strings to which they point will, of course, be disastrous.

Compiler implementation details

This section discusses implementation details of the MPW C Compiler. Most programs do not rely on these details and therefore yield the same results on various C compilers. However, if you want to write fast, compact programs, you may need to know details of how the MPW C Compiler is implemented. This section explains some of these details.

Size and byte alignment of variables

Variables of type `char`, and variables of type `enum` that require only a single byte of memory, are aligned in memory on byte boundaries. All other types are aligned on word boundaries (2 bytes): that is, they have even memory addresses. Note that the size of a `struct` is always a multiple of 2 bytes. This strategy makes memory access more efficient, because the MC68000 memory bus always reads at even addresses. By storing word- or multiple-word data types at even boundaries, the compiler guarantees that two-word data can be retrieved in a single memory access.

An enumerated type is allocated the amount of space required by the smallest predefined type that can represent all the literal values specified by the enumeration. The predefined types and their sizes are

`char (signed char)`	1 byte
`unsigned char`	1 byte
`short (signed short)`	2 bytes
`unsigned short`	2 bytes
`int (signed int )`	4 bytes
`unsigned int`	4 bytes

Signed and unsigned types have the same size and alignment.

Each new field of a structure begins on an even address except single-byte fields, which begin on an odd address if the previous field contained an odd number of bytes.

Byte ordering

The microprocessors used in the Macintosh computers (the Motorola MC68000, MC68020, and MC68030) store the least significant byte of a short or long integer at the highest memory address. This byte ordering is also used on IBM System/370 and Zilog Z8000 processors. The 6502 family of processors used in the Apple II family of computers, the Intel 8086 family, the DEC PDP-11 family, the DEC VAX, and the National Semiconductor NS16000 store the least significant byte at the lowest address. Programs that rely on the order of the bytes within short and long integers are not portable.

Variable allocation and register use

The compiler allocates automatic variables in registers whenever possible. Register variables are assigned to registers before other automatic variables are assigned. Enumeration, character, integer, and pointer variables qualify for register allocation unless their address is taken with the `&` operator. Static and global variables are not necessarily allocated in the order in which they are specified. (However, the order of fields within structures is preserved.) Automatic and static variables that are never used may not be allocated at all. Programs should not rely on the compiler's allocation algorithms.

Under the default compiler settings, floating-point variables are not allocated to registers. When the `mc68881` floating-point option is invoked, the compiler can place `extended` variables in the MC68881 registers.

Several data and address registers are available for use as automatic variables. The number of variables allocated to registers may exceed the total number of registers. Several variables whose useful lifetimes do not overlap may be assigned to the same register. Often all the eligible variables within a function reside in registers, rather than on the stack.

Registers D0, D1, D2, A0, and A1 (and FP0, FP1, FP2, and FP3 for `-mc68881`) are scratch registers that are not preserved by C functions. All other registers are preserved. Register A5 is the global frame pointer, register A6 is the local frame pointer, and register A7 is the stack pointer. Local stack frames are not necessarily created for simple functions, except when MacsBug symbol information is being generated.

(See Appendix D for information on how register use changed with MPW 3.0 C.)

Bit-fields

MPW C provides both signed and unsigned bit-fields: signed is the default. (This default is incompatible with MPW 2.0 C: see the section "Bit-fields" in Appendix D.)

Bit-fields are 1 to 32 bits long; they are packed in the order in which they appear in the structure, using the most significant bit of the destination first. A bit-field that does not fit into the space remaining in an `int` will be put into the next `int`. A bit-field can straddle a byte or word boundary, but not a long-word boundary. In a series of bit-fields, if one has the special width 0, then the next bit-field is guaranteed to be aligned on the next long-word boundary.

Bitwise operations

MPW 3.0 C stores signed integers as two's complement bit patterns, and the bitwise operations are implemented by the following MC68000 instructions:

```
~       NOT
<<      LSL
>>      ASR
&       AND
^       EOR
|       OR
```

Compiler limitations

Ordinarily, the total size of all declared global variables, static variables, and string constants cannot exceed 32K bytes. You can allocate large global arrays on the heap in order to avoid exceeding this limit. By using the `-m` compiler option, you can generate code that supports global data larger than 32K; however, this code is less efficient. If much of your global data consists of strings, you can optimize your program by using the `-b`, `-b2`, or `-b3` compiler options to move strings out of the global data area: using these options may require changes to your code if you pass pointers to string constants between segments.

The size of the largest function you can compile in MPW C is limited by the memory available for the compiler's internal data structures. The problem can be reduced by reducing the number of global declarations, compiling large functions separately, and rewriting large functions as two or more smaller functions. You can also make more memory available to the compiler by compiling without MacsBug installed.

The only limit to the number of case values in a `switch` statement is the stack and heap sizes.

Structure results

In MPW C, the caller passes in an address (as the first argument on the stack) to which to return a structure result. In some cases, this address will be the address of a structure to which the function is being assigned (for example, `foo = function();`) or will be the address of a temporary structure allocated in the calling procedure's stack frame. This method doesn't use global variables and it is reentrant. (This implementation is incompatible between MPW 2.0 C and 3.0: see the section "Structure Results," in Appendix D.)

Multiple external definitions

A distinction is made between function declarations and function definitions. According to the ANSI draft C standard, a program may contain multiple declarations for a function, but at most one definition. The MPW Linker uses the first complete definition it encounters and ignores any subsequent definitions.

Similarly, a distinction is made between data declarations (with keyword `extern`) and data definitions (without keyword `extern`). Again, the ANSI draft C standard allows multiple declarations for data, but at most one definition. For compatibility with existing programs, MPW C relaxes this restriction, allowing multiple definitions provided that at most one definition contains explicit initialization. If the linker finds multiple data definitions, it uses the first one it finds and ignores the rest.

The linker also supports K&R-style "common" blocks with the `-f` option, where the largest definition for an `extern` variable is used. Only one initialization is permitted.

Standard Apple Numerics Environment extensions

MPW C has built-in support for the Standard Apple Numerics Environment (SANE). SANE is introduced in this section and documented completely in the *Apple Numerics Manual,* second edition. The compiler supports SANE types and operations, and the C SANE Library provides functions that supplement the SANE facilities built into the MPW C language. Because the C SANE Library provides a number of functions, such as `exp`, that are found in the Standard C Library, the interfaces to these functions appear in Chapter 3. Additional functions appear in the *Apple Numerics Manual.*

The MPW C language and the C SANE Library together implement the IEEE Standard for Binary Floating-Point Arithmetic (754). SANE adds a data type to the IEEE types and provides basic functions for application development. MPW C recognizes the SANE data types (`float`, `double`, `comp`, and `long double`), and correctly handles NaNs (Not-a-Number) and infinities in comparisons and in ASCII-binary conversions.

Source programs that use only K&R C `float` and `double` types and K&R C operations are portable—they will compile and run in MPW C as well as in other C implementations.

MPW C uses SANE routines for all floating-point operations. To accommodate different hardware, you have the option of choosing software SANE or hardware SANE routines. By default, the compiler uses software SANE routines for all floating-point operations and conversions: such code runs on all Macintosh computers, but takes only partial advantage of numeric coprocessors like the MC68881. (Here the term *MC68881* also includes the MC68882.) If either the `-mc68881` or the `-elems881` option is invoked, the compiler uses MC68881 routines for some or all floating-point operations. (See Chapter 1.) Code produced with these options requires a numeric coprocessor like the MC68881 and takes full advantage of it.

Much of SANE is provided through the run-time library CSANELib.o (or CSANELib881.o) and its include file SANE.h. However, to use extended-precision arithmetic efficiently and effectively, and to handle IEEE NaNs and infinities, some extensions to K&R C are required, including the use of the `long double` data type.

The change from `double` to `long double` as the basic floating-point type is the most important numeric difference from K & R C. Because C was originally developed on the DEC PDP-11, the PDP-11 architecture is reflected in K&R C in the use of `float` and `double` as floating-point types, with `double` as the basic type: floating-point expressions are evaluated to `double`, anonymous variables are `double`, and floating-point parameters and function results are passed as type `double`. However, the low-level SANE arithmetic (as well as the Intel 8087 and Motorola MC68881 floating-point chips) evaluates arithmetic operations to the range and precision of the IEEE extended type; lower-precision arithmetic is actually slower. This type, called `extended`, has been in MPW C since version 1.0. Draft proposed ANSI C accommodates this type, defining a `long double` type, which can be more precise than the `double` type. Thus the `extended` type naturally replaces PDP-11 `double` type as the basic arithmetic type for computing purposes.

In MPW 3.0 C, the keywords `long double` and `extended` are synonymous. Both indicate the IEEE extended type: `long double` is the preferred form specified by the ANSI draft C standard; `extended` is the terminology of the IEEE standard and is retained for compatibility with earlier versions of MPW C.

- *Note:* The keyword `extended` means the IEEE extended type in all Apple languages and in many languages from other vendors. The keyword `long double` is more vaguely defined in the ANSI draft C standard: a conforming ANSI C compiler may interpret the keyword `long double` as the IEEE extended type, or as another long floating-point type, or even as a synonym for the `double` type. For this reason, developers interested in portability must be careful: their code may run when ported, but may produce different results.

The types `float` (IEEE single), `double`, and `comp` serve as space-saving storage types, just as `float` does in conventional C. The `comp` type, designed for storing large integral values like those used in accounting, is an Apple extension and is supported in other Apple languages.

The IEEE Standard specifies two special representations for its floating-point formats: NaNs (Not-a-Number) and infinities. MPW C expands the syntax for I/O to accommodate NaNs and infinities, and includes the treatment of NaNs in relationals as required by the IEEE Standard.

The SANE extensions to K&R C are backward compatible: programs written with only the `float` and `double` floating-point types and K&R C operations will compile and run without modification. SANE does not affect integer arithmetic.

For code that runs on a Macintosh II, the compiler can generate code containing direct MC68881 instructions instead of calls to the software SANE library for floating-point calculations. The results of arithmetic calculations, conversion between different binary formats, and comparisons are identical. An additional option (`-elems881`) is available when using the MC68881. This option directs the compiler to use direct 881 instructions for transcendental functions. This option provides faster, but slightly less accurate, results for these functions.

SANE comes in two flavors:

- **Software SANE**, which uses the Floating-Point Arithmetic Package and the the Transcendental Functions Package (described in Chapter 15 of *Inside Macintosh,* Volume II), runs on all machines and takes some advantage of numeric coprocessors.
- **Hardware SANE**, which makes direct MC68881 calls, requires a numeric coprocessor and uses it to full advantage.

Expressions

The SANE types—`float`, `double`, `comp`, and `long double`—can be mixed in expressions with one another and with integer types in the same manner that `float` and `double` can in K&R C. An expression consisting solely of a SANE-type variable, constant, or function is of type `long double`. An expression formed by subexpressions and an arithmetic operation is of type `long double` if either of its subexpressions is of that type. Expressions of type `long double` are evaluated with extended-precision SANE arithmetic, with conversions to type `long double` generated automatically as needed. Parentheses in `long double` expressions are honored. Initialization of external and static variables is done at compile time; all other evaluation of `long double` expressions is done at run time.

Comparisons involving a NaN

The result of a comparison involving a NaN operand is **unordered:** neither less than, greater than, nor equal. The usual set of comparison results—less than (<), greater than (>), and equal to (==)—is expanded to include unordered. For example, the negation of "*a* less than *b*" is not "*a* greater than or equal to *b*" but "(*a* greater than or equal to *b*) OR (*a* and *b* unordered)."

Functions

A numeric actual parameter passed by value is an expression and hence is of type `long double` or an integer type. All `long double` arguments are passed as `long double`. Similarly, all results of functions declared `float`, `double`, `comp`, or `long double` are returned as `long double`. For example, if you call a function in Math.h, it returns a `long double` even though the declaration reads `double`: this means that if you assign the result of such a call to a `long double` variable, it loses no accuracy before the assignment. Thus you get a more precise result, faster, than the draft standard might imply.

Numeric input/output

In addition to the usual syntax accepted for numeric input, the Standard C Library function `scanf` recognizes the string `"INF"` as infinity and the string `"NAN"` as a NaN. `"NAN"` may be followed by parentheses, which may contain an integer (a code indicating the NaN's origin). `"INF"` and `"NAN"` are optionally preceded by a sign and are case-insensitive. The `scanf` conversion specifiers for SANE types extend K&R C as follows: conversion characters `e`, `f`, and `g` indicate type `float`; `le`, `lf`, and `lg` indicate type `double`; `me`, `mf`, and `mg` indicate type `comp`; and `Le`, `Lf`, and `Lg` indicate type `long double`. (These specifiers are incompatible with MPW 2.0 C. See the section "The `scanf` and `printf` Functions," in Appendix D.)

The Standard C Library function `printf` writes infinities as `"INF"` and NaNs as `"NAN(`*ddd*`)"`, where *ddd* is the NaN code. `"INF"` and `"NAN(`*ddd*`)"` may be preceded by a minus sign.

Numerics environment

The numerics environment comprises the rounding direction, rounding precision, halt enables, and exception flags. IEEE-standard defaults—rounding to nearest, rounding to extended precision, and all halts disabled—are in effect for compile-time arithmetic (including decimal-to-binary conversion). Each program begins with these defaults and with all exception flags clear. Functions for managing the environment are included in the C SANE Library (CSANELib.o or CSANELib881.o). The compiler, in optimizing, will not change any part of the numerics environment (including the exception-flag settings, which are a side effect of arithmetic operations).

About the C SANE Library

The C SANE Library provides the basic tools for developing a wide range of applications. It includes the following features:

- logarithmic, exponential, and trigonometric functions
- financial functions
- random-number generation
- binary-decimal conversion
- numeric scanning and formatting
- environment control
- other functions required or recommended by the IEEE Standard

See Chapters 3 and 4 for the interface to the C SANE Library.

Programming with IEEE arithmetic

MPW C's automatic use of the `long double` type produces results that are generally better than those of most other C systems. For example, extended precision yields more accuracy, and extended range avoids unnecessary underflow and overflow of intermediate results. You can further exploit extended-precision arithmetic by declaring all floating-point temporary variables to be of type `long double`. This is both time-efficient and space-efficient, because it reduces the number of automatic conversions between types. External data should be stored in one of the three smaller SANE types (`float`, `double`, or `comp`), not only for economy but also because the `long double` format may vary between SANE implementations. As a general rule, use `float`, `double`, or `comp` data as program input; `long double` arithmetic for computations; and `float`, `double`, or `comp` data as program output.

In many instances, IEEE arithmetic allows simpler algorithms than were possible without IEEE arithmetic. The handling of infinities enlarges the domain of some formulas. For example, $1+1/x^2$ computes correctly even if x^2 overflows. While running with halts disabled (the default), a program will never crash because of a floating-point exception. Hence by monitoring exception flags, a program can test for exceptional cases after the fact. The alternative of screening out bad input is often infeasible, and sometimes impossible.

Chapter 3 The Standard C Library

THIS CHAPTER DESCRIBES THE STANDARD C Library provided with MPW C. The Standard C Library is a collection of basic routines that let you read and write files, examine and manipulate strings, perform data conversion, acquire and release memory, and perform mathematical operations.

The chapter contains descriptions of the library functions and macros grouped as in the corresponding header files. Under each header, the functions and macros are arranged by functionality. For example, the `fread` and `fwrite` functions are both found under the "The Standard Input/Output package—StdIO.h" header and the "Direct input/output" subheader. All of the function names and other identifiers used in Standard C Library routines are listed in Appendix C.

◆ *Note:* Remember that identifiers in C are case sensitive and should be spelled exactly as shown in the synopsis. ■

Contents

Chapter organization 73
Diagnostics—Assert.h 74
Character handling—CType.h 75
- Character testing—`isalpha`, `isupper`, `islower`, `isdigit`, `isxdigit`, `isalnum`, `isspace`, `ispunct`, `isprint`, `isgraph`, `iscntrl` 75
- Character case—`toupper`, `tolower`, `_toupper`, `_tolower`, `toascii` 77

Errors—ErrNo.h 78
Localization—Locale.h 82
- The `setlocale` function 82
- The `localeconv` function 83

Mathematics—Math.h 87
- Trigonometric functions—`sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2` 88
- Hyperbolic functions—`sinh`, `cosh`, `tanh` 89
- Exponential and logarithmic functions—`exp`, `log`, `log10`, `frexp`, `ldexp`, `modf` 89
- Power functions—`pow`, `sqrt` 90

Nearest integer, absolute value, and remainder functions—`floor`, `ceil`, `fmod`, `fabs` 91
Euclidean distance function—`hypot` 92
Floating-point-to-string conversion functions—`ecvt`, `fcvt` 92
Non-local jumps—SetJmp.h 93
Signal handling—Signal.h 95
Specifying signal handling—the `signal` function 96
Sending a signal—the `raise` function 97
The Standard Input/Output package—StdIO.h 98
Operations on files 102
Removing a file—`remove` 102
Renaming a file—`rename` 102
Creating a temporary file—`tmpfile` 103
Naming a temporary file—`tmpnam` 103
File access functions 104
Closing a file—`fclose`, `fflush` 104
Opening a file—`fopen`, `freopen`, `fdopen` 105
File buffering—`setbuf`, `setvbuf` 106
Formatted input/output functions 108
Printing formatted output—`printf`, `fprintf`, `sprintf`, `vprintf`, `vfprintf`, `vsprintf` 108
Converting formatted input—`scanf`, `fscanf`, `sscanf` 112
Character input/output functions 117
Getting character input—`getc`, `getchar`, `fgetc`, `getw` 117
Getting string input—`gets`, `fgets` 118
Character output—`putc`, `putchar`, `fputc`, `putw` 119
String output—`puts`, `fputs` 120
Undoing an input—`ungetc` 120
Direct input/output—`fread`, `fwrite` 121
File positioning—`fseek`, `rewind`, `ftell`, `fgetpos`, `fsetpos` 122
Error-handling—`feof`, `ferror`, `clearerr`, `perror`, `fileno` 123
Input/output control (non-ANSI extension)—IOCtl.h 124
The `ioctl` function 124
Low-level file I/O (non-ANSI extensions)—FCntl.h 126
Creating a file—`creat` 127
Opening a file—`open` 127
Closing a file—`close` 129
Removing a file—`unlink` 129
Reading from a file—`read` 130
Writing to a file—`write` 131

File positioning—lseek 132
Get file control and status information—faccess 133
Duplicate an open file descriptor—dup 136
Duplicate a file descriptor—fcntl 136
Variable arguments—StdArg.h 138
General utilities—StdLib.h 139
String-conversion functions 140
String-to-int conversion—atoi, atol 140
String-to-long-conversion—strtol, strtoul 140
String-to-float conversion—atof 141
String to double conversion—strtod 142
Random number functions—rand, srand 142
Memory-management functions—malloc, free, realloc, calloc 143
Communication with the environment 144
Program termination—abort 144
Install a function to be executed at program termination—atexit 145
Terminate the current application—exit 146
Access exported MPW shell variables—getenv 147
Communicate with the host system—system 148
Searching and sorting utilities 148
Search for an item in a sorted array—bsearch 148
Quicker sort—qsort 149
Integer arithmetic functions 150
Integer absolute value—abs, labs 150
Integer division—div, ldiv 150
Multibyte character functions—mblen, mbtowc, wctomb 151
Multibyte string functions—mbstowcs, wcstombs 152
String handling—String.h 153
String function conventions 154
Copying functions—memcpy, memccpy, memmove, strcpy, strncpy 154
Concatenation functions—strcat, strncat 155
Comparison functions—memcmp, strcmp, strncmp, strcoll, strxfrm 156
Search functions—memchr, strchr, strpbrk, strspn, strcspn, strstr, strtok 157
Miscellaneous functions—memset, strerror, strlen 159
Date and time—Time.h 159
Time-manipulation functions—clock, difftime, mktime, time 160
Time conversion functions—asctime, ctime, gmtime, localtime, strftime 161

Characteristics of floating types—Float.h 163
Sizes of integral types—Limits.h 164
Common definitions—StdDef.h 165

Chapter organization

The library routines under each header are documented as follows:

- *Synopsis* shows the code you need to add to your program when using these library routines and files you need to include at compile time.
- *Description* discusses the library routines and their input and output.
- *Return values* describes the values returned by the routines.
- *Diagnostics* describes error conditions.
- *Example* shows a sample use of the routine in some cases.
- *Note* contains remarks.
- *Warning* gives cautions.
- *See also* provides the names of other library routines or sections in this chapter related to the ones described in the current document.

The following header files declare the C functions described in this chapter. Your program must include the relevant header files.

Assert.h	SetJmp.h
Ctype.h	Signal.h
Errno.h	StdArgs.h
FCntl.h	StdDef.h
Float.h	StdLib.h
IOCtl.h	StdIO.h
Limits.h	String.h
Locale.h	Time.h
Math.h	

◆ *Note:* Header-file names in MPW C are not case sensitive. For example

```
#include <StdIO.h>
```

is equivalent to

```
#include <stdio.h>
```

Diagnostics—Assert.h

Synopsis

```
#include <Assert.h>
void assert (int expression);
```

Description

The `assert` macro allows your program to send diagnostic messages to the user, depending on the evaluation of `expression`. If `expression` is false, `assert` provides an error message, and then calls the `abort` function. The format for this message is

```
FILE myfile.c; Line 108 ##assertion failed: i<j
```

The `assert` macro can be activated using the NDEBUG macro. If NDEBUG is turned off, `assert` will evaluate `expression`.

Note

The message pointed to by `assert` is an executable MPW command. When the command is executed, it will open the source file containing the `assert` and highlight the `assert` statement.

Example

```
#include <assert.h>
main()
{
   int i,j;
   i = foo();
   j = 3;
   assert (i<100);       /* This depends on the compile-time */
                         /* setting of NDEBUG. If */
                         /* NDEBUG is defined,  the assert */
                         /* statement is not passed from the  */
                         /* preprocessor to the compiler */
   #undef NDEBUG         /* Turn off NDEBUG */
   #include <assert.h>
   assert (i<j);         /* This assert is turned on */
   #define NDEBUG        /* Turn on NDEBUG */
   #include <assert.h>
   j = foo();
   assert (j <100);      /* This assert is turned off */
}
```

See also

`abort`

Character handling—CType.h

These routines provide character testing and case-changing operations.

Character testing—`isalpha`, `isupper`, `islower`, `isdigit`, `isxdigit`, `isalnum`, `isspace`, `ispunct`, `isprint`, `isgraph`, `iscntrl`

These functions tell whether a given character is in one of several categories: alphabetical, uppercase, lowercase, digit, hex digit, white-space, punctuation mark, printing, graphic, or control. A separate function is provided for each category.

Synopsis

```
#include <CType.h>
int isalpha(int c);
int isupper(int c);
int islower(int c);
int isdigit(int c);
int isxdigit(int c);
int isalnum(int c);
int isspace(int c);
int ispunct(int c);
int isprint(int c);
int isgraph(int c);
int iscntrl(int c);
int isascii(int c);
```

Description

These macros return nonzero for true, zero for false, depending on the corresponding integer value of the given character. The `isascii` macro is defined on all integer values; the rest are defined only where `isascii` is true and on the single non-ASCII value EOF (–1). Table 3-1 shows when these macros return the value true.

■ **Table 3-1** Character-testing macros

Macro	Returns true if
`isascii`	`c` is an ASCII character code lower than128.
`isalpha`	`c` is a letter [A–Z] or [a–z].
`isupper`	`c` is an uppercase letter [A–Z].
`islower`	`c` is a lowercase letter [a–z].
`isdigit`	`c` is a digit [0–9].
`isxdigit`	`c` is a hexadecimal digit [0–9], [A–F], or [a–f].
`isalnum`	`c` is alphanumeric (letter or digit).
`isspace`	`c` is a space, tab, return, new line, vertical tab, or form feed.
`ispunct`	`c` is a punctuation character (neither control nor alphanumeric).
`isprint`	`c` is a printing character, space (32) through tilde (126).
`isgraph`	`c` is a printing character, similar to `isprint` except false for space.
`iscntrl`	`c` is a delete character (127) or an ordinary control character (less than 32).

Note

These macros do not support the Macintosh extended character set. For values outside the domain, the result is undefined.

Warning

If `c` is not in the domain of the function, the result is undefined.

See also

Character case

Character case—`toupper`, `tolower`, `_toupper`, `_tolower`, `toascii`

The first four of these routines change the case of a character. The `toascii` function converts any character to an ASCII character.

Synopsis

```
#include <CType.h>
int toupper(int c);
int tolower(int c);
int _toupper(int c);
int _tolower(int c);
int toascii(int c);
```

Description

The `toupper` and `tolower` functions have as their domain the set of ASCII characters (0 through 127) and the constant `EOF (-1)`. If parameter `c` to `toupper` represents a lowercase letter, the result is the corresponding uppercase letter. If parameter `c` to `tolower` represents an uppercase letter, the result is the corresponding lowercase letter. All other parameters in the domain are returned unchanged.

The `_toupper` and `_tolower` macros produce the same results as functions `toupper` and `tolower` but have restricted domains and are faster. Macro `_toupper` requires a lowercase letter as its parameter; its result is the corresponding uppercase letter. Macro `_tolower` requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Parameters outside the domain cause undefined results.

The `toascii` macro converts `c` by clearing all bits that are not part of a standard ASCII character. It is used for compatibility with other systems.

Note

These routines do not support the Macintosh extended character set (with values greater than 0x7F). For values outside the stated domain, the result is undefined.

Warning

The `_toupper` and `_tolower` macros must be used with care: parameters outside their respective domains cause undefined results. The `_toupper` macro requires a lowercase letter as its parameter: the `_tolower` macro requires an uppercase letter as its parameter.

See also

Character testing

Errors—ErrNo.h

Synopsis

```
#include <ErrNo.h>
extern int errno;
extern short MacOSErr;
```

Description

Many of the Standard C Library functions can return, in addition to their normal return values, a negative value indicating an error, typically –1. For example, a function returning a character as an `int` will indicate an error by returning –1, which is not a legitimate ASCII value. (See the descriptions of individual functions for details.) The external variable `errno` also provides an error number. Variable `errno` is not cleared on successful calls, so it should be tested only if the return value indicates an error.

The error name appears in brackets following the text in a library function description: for example,

"The next attempt to write a nonzero number of bytes will signal an error. `[ENOSPC]`"

Not all possible error numbers are listed for each library function because many errors are possible for most of the calls. Some UNIX operating system error numbers do not apply to Macintosh and are not documented in this manual. Some calls go to the Macintosh ROM and return a value in `MacOSErr` as well as the value in `errno`.

Table 3-2 is a list of the error numbers and their names as defined in the <ErrNo.h> file. Not all the error numbers listed here will be returned, but they are included in the header file for compatibility.

■ **Table 3-2** I/O errors

Value	Identifier	Message	Explanation
1	`EPERM`	*No permission match*	An attempt was made to modify a file in some way forbidden except to its creator.
2	`ENOENT`	*No such file or directory*	A file whose filename is specified does not exist, or one of the directories in a pathname does not exist.
3	`ENORSRC`	*Resource not found*	A required resource was not found. This error applies to `faccess` calls that return tab, font, or print record information.
4	`EINTR`	*System service interrupted*	A requested system call cannot be completed. This error may occur if a request to rename a file is unsuccessful.
5	`EIO`	*I/O error*	Some physical I/O error has occurred. This error may in some cases be signaled on a call following the one to which it actually applies.
6	`ENXIO`	*No such device or address*	I/O on a special file refers to a subdevice that does not exist, or the I/O is beyond the limits of the device. This error may also occur when, for example, no disk is present in a drive.

(continued)

■ **Table 3-2** (continued) I/O errors

Value	Identifier	Message	Explanation
7	E2BIG	*Insufficient space for return argument*	The data to be returned is too large for the space allocated to receive it.
9	EBADF	*Bad file number*	Either a file descriptor does not refer to an open file, or a read (or write) request is made to a file that is open only for writing (or reading).
12	ENOMEM	*Not enough space*	The system ran out of memory while the library call was executing.
13	EACCES	*Permission denied*	An attempt was made to access a file in a way forbidden by the protection system.
14	EFAULT	*Illegal filename*	A filename or volume name was too long or otherwise illegal.
15	ENOTBLK	*Block device required*	A non-block file was used when a block device was required.
16	EBUSY	*Device or resource busy*	An attempt was made to mount a volume that was already mounted, or to delete a locked file.
17	EEXIST	*File exists*	An existing file was mentioned in an inappropriate context; for example, `open(file, O_CREAT+O_EXCL)`.
18	EXDEV	*Cross-device link*	A link to a file on another device was attempted.
19	ENODEV	*No such device*	An attempt was made to apply an inappropriate system call to a device; for example, read a write-only device.
20	ENOTDIR	*Not a directory*	An object that is not a directory was specified where a directory is required; for example, in a path prefix.
21	EISDIR	*Is a directory*	An attempt was made to write on a directory.
22	EINVAL	*Invalid parameter*	Some invalid parameter was provided to a library function.

(continued)

■ **Table 3-2** (continued) I/O errors

Value	Identifier	Message	Explanation
23	ENFILE	*File table overflow*	The system's table of open files is full, so temporarily a call to `open` cannot be accepted.
24	EMFILE	*Too many open files*	The system cannot allocate memory to record another open file.
25	ENOTTY	*Not a typewriter*	The specified file isn't a character file.
26	ETXTBSY	*Text file busy*	An attempt was made to open a file that was already open for writing.
27	EFBIG	*File too large*	The size of a file was larger than the maximum file size.
28	ENOSPC	*No space left on device*	During a write to a file, there is no free space left on the device.
29	ESPIPE	*Illegal seek*	An `lseek` was issued incorrectly.
30	EROFS	*Read-only file system*	An attempt to modify a file or directory was made on a device mounted for read-only access.
31	EMLINK	*Too many links*	An attempt to delete an open file was made.
33	EDOM	*Math arg out of domain of func*	The argument of a math function is outside the domain of the function.
34	ERANGE	*Math result not representable*	The value of a math function can't be represented within the machine's precision.

Note

Calls that interface to the Macintosh I/O system (such as `open`, `close`, `read`, `write`, and `ioctl`) can set the external variable `MacOSErr` as well as `errno` on errors. This section documents the `errno` values. The equivalent Macintosh ROM error-return values set in `MacOSErr` are documented in the System Error Handler chapter of *Inside Macintosh*.

See also

`hypot, signal, perror, creat, open, close, read, write, ioctl`

Localization—Locale.h

The header <Locale.h> declares several macros and the setlocale and localeconv functions, used for program localization. The various locale categories are defined as macros as listed in Table 3-3:

■ **Table 3-3** Setlocale categories

Macro name	Value	Affects
NULL	0	
LC_ALL	1	the program's entire locale
LC_COLLATE	2	behavior of the strcoll and strxfrm functions
LC_CTYPE	3	character handling and multibyte functions
LC_MONETARY	4	monetary formatting returned by localeconv function
LC_NUMERIC	5	decimal-point character for formatted I/O and string conversion, and nonmonetary formatting returned by localeconv function
LC_TIME	6	the behavior of the strftime function

You can specify one of these categories as the first argument when using the setlocale function.

See also

Formatted input/output functions, multibyte functions, string conversion functions

The setlocale function

Synopsis

```
#include <Locale.h>
char *setlocale (int category, const char *locale);
```

Description

The `setlocale` function provides control over certain categories of a program's locale. You can use the `setlocale` function to change or query the program's entire current locale or portions of the current locale. The `category` argument is one of the macros listed in the previous section. The `LC_ALL` macro specifies the program's entire locale, while the other macros specify a portion of the locale.

The `locale` argument is a constant. You can specify an empty string constant (`""`) to use your system file's setting for the native environment: in MPW 3.0 C, specifying the `""` locale is the same as specifying the `"US"` locale.

You can use the null string constant (`"\0"`) for `locale` to find the name of the current native environment for a given `category`.

For compatibility with the draft ANSI standard, you can also supply a value of `"C"` as the locale argument.

Return values

If you used a valid `locale` corresponding to a `category`, the `setlocale` function returns a pointer to the name of the locale for that `category`. If `setlocale` fails to locate the locale name, it returns a null pointer.

The `localeconv` function

Synopsis

```
#include <Locale.h>

struct lconv {
   char *decimal_point;
   char *thousands_sep;
   char *grouping;
   char *int_curr_symbol;
   char *currency_symbol;
   char *mon_decimal_point;
   char *mon_thousands_sep;
   char *mon_grouping;
   char *positive_sign;
   char *negative_sign;
   char int_frac_digits;
   char frac_digits;
   char p_cs_precedes;
   char p_sep_by_spaces;
```

```
    char n_cs_precedes;
    char n_sep_by_spaces;
    char p_sign_posn;
    char n_sign_posn;
}
struct lconv *localeconv (void);
```

Description

The `localeconv` function sets numeric formatting values for type `struct lconv`. The components of `lconv` can be set according to the rules of the current locale.

The `lconv` fields of type `char *` are strings. Any of these structure members (except `decimal_point`) can point to `""`, indicating either that the value isn't available in the current locale or that its length equals zero. The `lconv` fields of type `char` are nonnegative numbers. Any of these structure members can have the value `CHAR_MAX`, indicating that its value isn't available in the current locale. The `lconv` fields are listed below:

`char *decimal_point`	The decimal-point character used to format nonmonetary quantities.
`char *thousands_sep`	The character used to separate groups of digits to the left of the decimal-point character in formatted nonmonetary quantities.
`char *grouping`	A string whose elements indicate the size of each group of digits in formatted nonmonetary quantities.
`char *int_curr_symbol`	The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in *ISO 4217 Codes for the Representation of Currency and Funds*. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.
`char *currency_symbol`	For the current locale, this character denotes the local currency symbol.
`char *mon_decimal_point`	This character is the decimal point for formatting monetary quantities.
`char *mon_thousands_sep`	This character is used with formatted monetary quantities as the separator for groups of digits to the left of the decimal point.

`char *mon_grouping`	This string includes elements indicating the size of each group of digits in formatted monetary quantities.
`char *positive_sign`	This string indicates a nonnegative-valued formatted monetary quantity.
`char *negative_sign`	This string indicates a negative-valued formatted monetary quantity.
`char int_frac_digits`	For an internationally formatted monetary quantity, this element indicates the number of fractional digits (those to the right of the decimal point) to be displayed.
`char frac_digits`	For a formatted monetary quantity, this element indicates the number of fractional digits (those to the right of the decimal point) to be displayed.
`char p_cs_precedes`	This value is set to 1 if the `currency_symbol` precedes the value, or 0 if the `currency_symbol` succeeds the value for a nonnegative formatted monetary quantity.
`char p_sep_by_space`	This value is set to 1 if the `currency_symbol` is separated by a space from the value, or to 0 if the `currency_symbol` is not separated by a space from the value, for a nonnegative formatted monetary quantity.
`char n_cs_precedes`	This value is set to 1 if the `currency_symbol` precedes the value, or to 0 if the `currency_symbol` succeeds the value, for a negative formatted monetary quantity.
`char n_sep_by_space`	This value is set to 1 if the `currency_symbol` is separated by a space from the value, or to 0 if the `currency_symbol` is not separated by a space from the value, for a negative formatted monetary quantity.
`char p_sign_posn`	This character is set to a value that indicates how the `positive_sign` is positioned for a nonnegative formatted monetary quantity.
`char n_sign_posn`	This character is set to a value that indicates how the `negative_sign` is positioned for a negative formatted monetary quantity.

The following constants affect the interpretation of the `grouping` and `mon_grouping` fields:

`CHAR_MAX`	Performs no further grouping.
`0`	Uses the previous element repeatedly for the remainder of the digits.
other	Examines the next element to determine the size of the next group of digits to the left of the current group. The value of *other* equals the number of digits in the current group.

The following constants affect the interpretation of the `p_sign_posn` and `n_sign_posn` elements:

`0`	The quantity and `currency_symbol` are surrounded by parentheses.
`1`	The sign string occurs before the quantity and `currency_symbol`.
`2`	The sign string occurs after the quantity and `currency_symbol`.
`3`	The sign string occurs immediately before the `currency_symbol`.
`4`	The sign string occurs immediately after the `currency_symbol`.

Values within the `lconv` struct will only change if the user's program explicitly changes them.

Return values

The `localeconv` function returns a pointer to the filled-in structure `lconv`. The `lconv` structure fields shouldn't be modified by the program, so that the contents will remain the same until the next `localeconv` call. Subsequent calls to `localeconv` can overwrite fields within the structure. Calling `setlocale` by using one of the categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC` may also overwrite the contents of the `lconv` structure.

Example

The following examples show the C and US formatting for monetary quantities:

Element	"C"	"US"
char *decimal_point;	"."	"."
char *thousands_sep;	""	","
char *grouping;	""	""
char *int_curr_symbol;	""	""
char *currency_symbol;	""	"$"
char *mon_decimal_point;	""	"."
char *mon_thousands_sep;	""	","
char *mon_grouping;	""	""
char *positive_sign;	""	""
char *negative_sign;	""	""
char int_frac_digits;	CHAR_MAX	CHAR_MAX
char frac_digits;	CHAR_MAX	CHAR_MAX
char p_cs_precedes;	CHAR_MAX	CHAR_MAX
char p_sep_by_spaces;	CHAR_MAX	CHAR_MAX
char n_cs_precedes;	CHAR_MAX	CHAR_MAX
char n_sep_by_spaces;	CHAR_MAX	CHAR_MAX
char p_sign_posn;	CHAR_MAX	CHAR_MAX
char n_sign_posn;	CHAR_MAX	CHAR_MAX

See also

Formatted input/output functions, multibyte functions, string conversion functions

Mathematics—Math.h

This header file provides mathematical functions: trigonometric, hyperbolic, exponential and logarithmic, floating-point manipulation, power, conversion, and miscellaneous functions. It provides all the mathematical functions specified in the ANSI draft C standard, as well as additional functions in the C SANE library.

Trigonometric functions—sin, cos, tan, asin, acos, atan, atan2

Synopsis

```
#include <Math.h>
extended sin(extended x);
extended cos(extended x);
extended tan(extended x);
extended asin(extended x);
extended acos(extended x);
extended atan(extended x);
extended atan2(extended x, extended y);
```

Description

The `sin`, `cos`, and `tan` functions return, respectively, the sine, cosine, and tangent of their argument, which is in radians.

The `asin` function returns the arcsine of `x`, in the range $-\pi/2$ to $\pi/2$.

The `acos` function returns the arccosine of `x`, in the range 0 to π.

The `atan` function returns the arctangent of `x`, in the range $-\pi/2$ to $\pi/2$.

The `atan2` function returns the arctangent of `y/x`, in the range $-\pi$ to π, using the signs of both arguments to determine the quadrant of the return value.

For special cases, these functions return a NaN or infinity as appropriate.

Diagnostics

These functions honor the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

Note

The `sin`, `cos`, and `tan` functions have periods based on the nearest extended-precision representation of mathematical π. Hence these functions diverge from their mathematical counterparts as their argument becomes far from zero.

See also

Apple Numerics Manual, 2nd edition

Hyperbolic functions—`sinh`, `cosh`, `tanh`

Synopsis

```
#include <Math.h>
extended sinh(extended x);
extended cosh(extended x);
extended tanh(extended x);
```

Description

The `sinh`, `cosh`, and `tanh` functions return, respectively, the hyberbolic sine, cosine, and tangent of their parameter.

Diagnostics

The `sinh`, `cosh`, and `tanh` functions honor the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

See also

`exp`

Apple Numerics Manual, 2nd edition

Exponential and logarithmic functions—`exp`, `log`, `log10`, `frexp`, `ldexp`, `modf`

Synopsis

```
#include <Math.h>
extended exp(extended x);
extended log(extended x);
extended log10(extended x);
extended frexp(extended value, int *exp);
extended ldexp(extended x, int exp);
extended modf(extended value, extended *iptr);
```

Description

The `exp(x)` function returns e^x, where e is the natural logarithm base.

The `log(x)` function returns the natural logarithm of `x`, $\log_e x$.

The `log10(x)` function returns the base 10 logarithm of x.

The `frexp` function returns the mantissa of an extended `value` and stores the exponent pointed to by `exp`. Every nonzero number can be written uniquely as $x * 2^n$, where the mantissa (fraction) x is in the range $0.5 \leq |x| < 1.0$ and the exponent n is an integer. Note that the mantissa here differs from the significand described in the *Apple Numerics Manual,* 2nd edition, whose normal values are in the range $1.0 \leq |x| < 2.0$.

The `ldexp` function returns the quantity `x` * 2^{exp}.

The `modf` function returns the signed fractional part of `value` and stores the integral part indirectly in the location pointed to by `iptr`.

For special cases, these functions return a NaN or signed infinity as appropriate.

Diagnostics

The `exp`, `log`, `log10`, and `ldexp` functions honor the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

See also

`hypot, sinh`

Apple Numerics Manual, 2nd edition

Power functions—`pow, sqrt`

Synopsis

```
#include <Math.h>
extended pow(extended x, extended y);
extended sqrt(extended x);
```

Description

The MPW C macro `pow(x,y)` is equivalent to the SANE function `power(x,y)`, which returns x^y.

The `sqrt(x)` function returns the square root of `x`. For special cases, this function returns a NaN or signed infinity as appropriate.

Diagnostics

These functions honor the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

See also

`hypot, sinh`

Apple Numerics Manual, 2nd edition

Nearest integer, absolute value, and remainder functions—`floor`, `ceil`, `fmod`, `fabs`

Synopsis

```
#include <Math.h>
extended floor(extended x);
extended ceil(extended x);
extended fmod(extended x, extended y);
extended fabs(extended x);
```

Description

The `floor(x)` function returns the largest integer (as an extended-precision number) not greater than `x`.

The `ceil(x)` function returns the smallest integer not less than `x`.

Whenever possible, the `fmod(x, y)` function returns the number *f* with the same sign as `x`, such that `x` = *i* `y` + *f* for some integer *i*, and | *f* | < |`y`|. If `y` is 0, `fmod` returns a NaN.

The `fabs(x)` function returns |`x`|, the absolute value of `x`.

See also

`abs`

Apple Numerics Manual, 2nd edition

Euclidean distance function—`hypot`

Synopsis

```
#include <Math.h>
extended hypot(extended x, extended y);
```

Description

The `hypot` function returns the square root of the sum of two squared numbers:
`sqrt(x*x + y*y)`

The `hypot` function includes error handling for those cases when the correct value would overflow. If the function result overflows, `hypot` returns `HUGE_VAL` with the correct sign, and sets `errno` to `ERANGE`.

Diagnostics

The function `hypot` honors the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

See also

`exp`

Apple Numerics Manual, 2nd edition

Floating-point-to-string conversion functions—`ecvt`, `fcvt`

Synopsis

```
#include <Math.h>
char *ecvt(extended value, int ndigit, int *decpt, int *sign);
char *fcvt(extended value, int ndigit, int *decpt, int *sign);
```

Description

The `ecvt` function converts `value` to a null-terminated string of `ndigit` digits and returns a pointer to this string as the function result. The low-order digit is rounded.

The decimal point is not included in the returned string. The position of the decimal point is indicated by `decpt`, which indirectly stores the position of the decimal point relative to the returned string. If the `int` pointed to by `decpt` is negative, the decimal point lies to the left of the returned string. For example, if the string is `"12345"` and `decpt` points to an `int` of 3, the value of the string is 123.45; if `decpt` points to –3, the value of the string is .00012345.

If the sign of the converted value is negative, the `int` pointed to by `sign` is nonzero; otherwise it is zero.

The `fcvt` function provides fixed-point output in the style of Fortran F-format output. The function `fcvt` differs from `ecvt` in its interpretation of `ndigit`:

- In `fcvt`, `ndigit` specifies the number of digits to the right of the decimal point.
- In `ecvt`, `ndigit` specifies the number of digits in the string.

◆ *Note:* The string pointed to by the function result is static data whose contents are overwritten by each call. To preserve the value, copy it before calling the function again.

See also

`printf`

Apple Numerics Manual, 2nd edition

Non-local jumps—SetJmp.h

Synopsis

```
#include <SetJmp.h>
typedef int jmp_buf[12];

int setjmp (jmp_buf env);  /* macro only */
void longjmp (jmp_buf env, int val);
```

Description

These functions let you escape from an error or interrupt encountered in a low-level subroutine of your program.

The `setjmp` function saves its stack environment in `env` for use by `longjmp` calls that occur later in the same calling chain. (See the examples). The `setjmp` function returns the value 0.

When the `longjmp` function is called with the same `env`, it restores the `env` saved by the most recent `setjmp` call with the same `env`, and the program executes the code at the `setjmp` location. The program continues as if the preceding call to `setjmp` had returned the value `val`.

The `longjmp` function cannot cause `setjmp` to return the value 0. If `longjmp` is invoked with value `val` equal to 0, `setjmp` returns 1. Data values will be those in effect at the time `longjmp` was called, except for register variables. (See "Warning.")

Examples

Here are two ways to do the same thing. The first method presents `setjmp` error handling first, then shows the other routines:

```
if (setjmp(&env) != 0) { /* env is a global variable */
   /* error handling code executed for the else clause*/
} else {
   call (something);
   /* some call chain that includes a longjmp */
}  /* end if */
```

The second method presents the call chain first, then shows the error handling:

```
if (setjmp(&env) == 0 ){ /* env is a global variable */
   /* some call chain that includes a longjmp */
   call (something);
} else {
   /* error handling code executed after "return" from longjmp */
}  /* end if */
```

You must call `longjmp` later than `setjmp` in the call chain to produce the desired behavior, which is to get `setjmp` to return a value other than 0. If you exit from the outer routine, where the call to `setjmp` occurred, then the call to `longjmp` won't work properly.

Warning

If `longjmp` is called without a previous call to `setjmp` with the same `env`, or if the function that contained the call to `setjmp` returned before the call to `longjmp` occurred, results are unpredictable. (See the examples.)

After a `longjmp`, variables that happen to be assigned to registers are restored to the values they had before the call to `setjmp`, instead of those they had at the time `longjmp` was called. To avoid this problem, declare "important" variables as `volatile`, to prohibit their use as register variables.

See also

`abort, signal`

Signal handling—Signal.h

C programs that handle software interrupts—known as **signals**—should use the functions described below for signal handling under the MPW Shell. A signal is similar to a hardware interrupt: its invocation can cause program control to be temporarily diverted from its normal execution sequence. The difference between a hardware interrupt and a signal is that signal events reflect a change in program state rather than hardware state. Examples of signal events are stack overflow, heap overflow, software floating-point exceptions and Command-period interrupts.

Interactive interrupt monitoring is available only for tools running under the MPW Shell. Currently the only software interrupt provided is the interactive interrupt signal, Command-period, represented by the value `SIGINT`. Stand-alone applications that run under the Macintosh Finder™ must provide their own interactive interrupt mechanism. Under MPW 3.0, signals can be set with the `signal` function, and sent with the `raise` function. The default action of any signal raised is to close all open files, execute any exit procedures installed with `atexit`, and terminate the program. No signal-handling calls are required to execute a normal termination on receipt of a signal.

The header Signal.h defines the type

```
typedef int sig_atomic_t;
```

This is the integral type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

The header Signal.h also defines the macros

```
#define SIG_DFL 1
#define SIG_ERR -1
#define SIG_IGN 0
```

These expand to distinct constant expressions that are type-compatible with the second argument to and the return value of the `signal` function. These expressions compare unequal to the address of any function that can be declared.

Specifying signal handling—the `signal` function

Synopsis

```
#include <Signal.h>
void (*signal (int sig, void (*func) (int))) (int);
```

Description

The `signal` function returns the signal number `sig`, and takes action depending on its value. If the macro value passed in `func` is `SIG_DFL`, default signal handling occurs. With a value of `SIG_IGN`, the signal is ignored. In all other cases, `func` is a pointer to a signal handler that is called when the signal is received.

Upon entry to a user-supplied signal handler, all signals are temporarily suspended; therefore, the handler is not required to lock out recursive or nested calls to signal handlers. The signal state is restored upon normal return from the signal handler. Signals cannot be raised while executing ROM code or an MPW tool. If a signal event occurs while executing outside the tool, the signal state is set and the signal handler is executed as soon as program control returns to the tool. Because a signal can interrupt the tool at any point, there is no protection against heap corruption if a signal handler executes calls that modify the state of the heap. Because most buffered I/O potentially modifies the heap, `printf` and similar calls are not recommended in signal handlers. Even then, the caller must be careful of interaction between `exit` and `atexit` procedures.

Return values

The `signal` function normally returns a value for `func` corresponding to the last call to signal using the specified signal `sig`. If the signal request can't be honored, the `signal` function returns a value of `SIG_ERR` and places a positive value in `errno`.

Example

```
#include <Signal.h>
static sig_atomic_t flag;

void myhandler (int sig)
{
   flag = 1;
   (void) signal (sig, myhandler);  /* reinstall this signal handler */
}

void foo()
{
void (*sighandler)(int);
   flag = 0;
   sighandler = signal (SIGINT, myhandler);
   if (sighandler == SIG_ERR) {  /*error handling code here */}
   /* start of uninterruptable code here */
      ...
      ...
      ...
   /*end of uninterruptable code */
   if (flag == 1) {
   /*do interrupt processing here */
   }
(void) signal (SIGINT, sighandler);
}
```

Note

The default signal handler is installed immediately before a user-supplied signal handler is called. If you want your signal handler to remain installed, you must reinstall it. You may do so within the signal handler itself.

See also

`abort, setjmp`

Sending a signal—the `raise` function

Synopsis

```
#include <Signal.h>
int raise (int sig); /* new routine to send a signal */
```

Description

The `raise` function can be used to send a signal, specified by using the `sig` parameter, to the currently executing program.

Return values

The `raise` function returns either a zero value if the signal was successful, or a nonzero value if the signal was unsuccessful.

See also

`abort, setjmp`

The Standard Input/Output package—StdIO.h

Synopsis

```
#include <StdIO.h>
#define size_t unsigned int
#define NULL 0

/* The basic data structure for a stream is the FILE.*/

typedef struct {
   int _cnt;
   unsigned char *_ptr;
   unsigned char *_base;
   unsigned char *_end;
   unsigned short _size;
   unsigned short _flag;
   unsigned short _file;
} FILE;

FILE *stdin, *stdout, *stderr;
typedef long fpos_t;
```

Description

The Standard I/O package constitutes an efficient user-level I/O buffering scheme. The inline macros `getc` and `putc` handle characters quickly.

The following macros and higher-level routines all use `getc` and `putc`:

```
getchar   putchar   fgetc   fgets
fprintf   fputc     fputs   fread
fscanf    fwrite    gets    getw
printf    puts      putw    scanf
```

Therefore calls to these macros and functions can be freely intermixed.

The constants and the following functions are implemented as macros:

```
getc   getchar   putc       putchar
feof   ferror    clearerr   fileno
```

Avoid redeclaration of these names.

Any program that uses the Standard I/O package must include the <StdIO.h> header file of macro definitions. The functions, macros, and constants used in the Standard I/O package are declared in the header file and need no further declaration.

A stream is a file with associated buffering and is declared to be a pointer to a `FILE` variable. The functions `fopen`, `freopen`, and `fdopen` return this pointer. The information in the `FILE` variable includes

- the file access—read or write
- the file descriptor as returned by `open`, `creat`, `dup`, or `fcntl`
- the buffer size and location
- the buffer style (unbuffered, line buffered, or file buffered)

Standard I/O buffering

Output streams, with the exception of the standard error stream `stderr`, are by default file buffered if the output refers to a file. File `stderr` is by default line buffered. When an output stream is **unbuffered,** it is queued for writing on the destination file or window as soon as written; when it is **file buffered,** many characters are saved up and written as a block; when it is **line buffered,** each line of output is queued for writing as soon as the line is completed (that is, as soon as a newline character is written). The function `setvbuf` may be used to change the stream's buffering strategy.

Normally, there are three open streams with constant pointers declared in the <StdIO.h> header file and associated with the standard open files, as shown in Table 3-4:

■ **Table 3-4** Standard open files

FILE variable	Fildes	Description	Buffer style
stdin	0	standard input file	file buffered
stdout	1	standard output file	file buffered
stderr	2	standard error file	line buffered

Buffer initialization

The `FILE` variable returned by `fopen`, `freopen`, or `fdopen` has an initial buffer size of 0 and a `NULL` buffer pointer. The buffer size is set and the buffer allocated by a call to `setbuf`, `setvbuf`, or the first I/O operation on the stream, whichever comes first. Buffer initialization is done using the following algorithm:

1. If `_IONBF` (no buffering) was set by a call to `setvbuf`, initialization steps 2 and 3 are skipped. The buffer size remains 0 and the buffer pointer remains `NULL`.
2. Checks the access-mode word for `_IOLBF` (line buffering). This bit is usually set only in the predefined `FILE stderr`, but a call to `setvbuf` can set it for any file. If line buffering is set, the buffer size is set to `LBUFSIZ` (100). If line buffering is not set, `ioctl` is called with an `FIOBUFSIZE` request and the buffer size is set to the returned value or to `BUFSIZ` (1024) if no value is returned.
3. If the buffer pointer is `NULL`, a request is made for a buffer whose size was determined in step 2; the buffer pointer is set to point to the newly allocated buffer. If the requested size cannot be allocated, attempts are made to allocate `BUFSIZ` or `LBUFSIZ` if these are smaller than the requested size. If all requests fail, the buffer pointer remains `NULL` and the `_IONBF` (no buffering) bit is set.
4. Function `ioctl` is called with an `FIOINTERACTIVE` request; if it returns `true`, the `_IOSYNC` bit is set in the access-mode word. This is done for all `FILE` variables, regardless of their buffering style and size. (The `_IOSYNC` bit is described in the next section.)

The `setvbuf` function lets you specify values for buffer size, buffer pointer, and access-mode word other than the default values of 0, `NULL`, and 0, respectively. The `setvbuf` function must be called before the first I/O operation occurs, so that the buffer initialization procedure described above receives the values you specify instead of the default values.

Buffered I/O

On each write request, the bytes are transferred to the buffer and an internal counter is set to account for the number of bytes in the buffer. If `_IOLBF` is set and a newline character is encountered while transferring bytes to the buffer, the buffer is flushed (written immediately) and the transfer continues at the beginning of the buffer. This process continues until the write-request count is satisfied or a write error occurs.

On each read request, the `_IOSYNC` bit in the access-mode word is checked. If `_IOSYNC` is on, all current `FILE` variables that have `_IOSYNC` on and are open for writing are flushed. In other words, a read from an interactive `FILE` variable flushes all interactive output files before reading. This flushing ensures that any prompts, I/O in a window, or other visual feedback is displayed before the read is initiated. Then, if the internal counter is 0, an entire buffer is read into memory if possible. (For the console device, less than a buffer's worth is likely to be read.) The bytes required to satisfy the read request are transferred, going back to the device for more if necessary, and an internal pointer is advanced if any bytes remain unread.

When the Standard I/O package is used, Standard I/O cleanup is performed just before termination of the application. Any normal return, including a call to `exit`, causes Standard I/O cleanup, which consists of a call to `fclose` for every open `FILE` stream.

Diagnostics

Most integer functions that deal with streams return the integer constant `EOF` (–1) upon reaching the end-of-file or if an error occurs. See the descriptions of the individual functions for details.

Note

Do not use a file descriptor (0, 1, or 2) where a `FILE` variable (`stdin`, `stdout`, or `stderr`) is required.

The file <StdIO.h> includes definitions other than those described above, but their use is not recommended.

Invalid stream pointers cause serious errors, possibly including program termination. Individual function descriptions describe the possible error conditions.

The `fpos_t` type can express any position in a file. A file's end-of-file marker has type `fpos_t`.

See also

`close, exit, fclose, ferror, fopen, fread, fseek, getc, gets, lseek, atexit, open, printf, putc, puts, read, scanf, setbuf, ungetc, write`

Operations on files

The Standard I/O package provides functions to remove and rename files, and to create and name temporary files.

Removing a file—`remove`

Synopsis

```
#include <StdIO.h>
int remove(const char *filename);
```

Description

The `remove` function removes the named file, specified by `filename`. If you subsequently attempt to use `open` or `reopen` with that filename, the operation will fail.

Return values

The `remove` function returns a nonzero value if it fails, and a zero value if it succeeds.

See also

`fopen`

Renaming a file—`rename`

Synopsis

```
#include <StdIO.h>
int rename(const char *old, const char *new);
```

Description

The `rename` function changes the name of a file, specified by `old`. The file is renamed using the name contained in the string pointed to by `new`.

Return values

The `rename` function returns a nonzero value if it fails, so if the file existed previously it is still known by its original name. It returns a zero value if the renaming operation succeeds.

See also

`fopen`

Creating a temporary file—`tmpfile`

Synopsis

```
#include <StdIO.h>
FILE *tmpfile(void);
```

Description

The `tmpfile` function creates and opens a temporary binary file. This file is automatically removed when it is closed or when the program terminates.

Return values

The `tmpfile` function returns a null pointer if the temporary file cannot be created. If the `tmpfile` operation succeeds, it returns a pointer to the stream of the temporary file.

See also

`fopen`

Naming a temporary file—`tmpnam`

Synopsis

```
#include <StdIO.h>
char *tmpnam(char *s);
```

Description

The `tmpnam` function creates a string that can be used as a valid filename. This string is not the same as the name of any existing file. The `tmpnam` function generates a different filename each time it is called, and can be called up to `TMP_MAX` times during program execution.

Return values

With an argument of `tmpnam(NULL)`, the `tmpnam` function places its result in an internal static object and returns a pointer to that string. Subsequent calls may modify this string. With an argument of `tmpnam(s)`, the variable `s` points to an array of at least `L_tmpnam` chars; the `tmpnam` function then places its result in that array before returning the argument as its value.

See also

`fopen`

File access functions

The Standard I/O package provides functions to open and close files and to let you provide buffering for a file.

Closing a file—`fclose`, `fflush`

Synopsis

```
#include <StdIO.h>
int fclose(FILE *stream);
int fflush(FILE *stream);
```

Description

The `fclose` function closes a file that was opened by `fopen`, `freopen`, or `fdopen`. The function `fclose` causes any buffered data for `stream` to be written out, and the buffer (if one was allocated by the system) to be released; `fclose` then calls `close` to close the file descriptor associated with `stream`. The value of the parameter `stream` cannot be used unless reassigned with `fopen`, `fdopen`, or `freopen`.

The `fclose` function fails, returning `ENOENT`, if the file descriptor associated with `stream` is already closed.

The `fclose` function is performed automatically for all open `FILE` streams upon calling `exit`.

The `fflush` function causes any buffered data for `stream` to be written out; `stream` remains open.

Return value

These functions return 0 for success or –1 if an error was detected (such as trying to write to a file that has not been opened for writing).

See also

`close, exit, fclose, fopen, setbuf, stdio`

Opening a file—`fopen, freopen, fdopen`

Synopsis

```
#include <StdIO.h>
FILE *fopen(const char *filename, const char *mode);
FILE *freopen(const char *filename, const char *mode, FILE *stream);
FILE *fdopen(int fildes, char *type);
```

Description

The `fopen` function opens the file named by `filename` and associates a stream with it.

The `filename` parameter points to a character string that contains the name of the file to be opened.

The `mode` parameter points to a character string beginning with one of the values in the first column in the following table. The columns "Open Mode Used" and "Description" explain how type is used. For more information, see `open`.

Value	Open mode used	Description
`r`	`O_RDONLY`	Open for reading only.
`w`	`O_WRONLY\|O_CREAT\|O_TRUNC`	Truncate or create for writing.
`a`	`O_WRONLY\|O_APPEND`	Append: open for writing at end-of-file, or create for writing.
`r+`	`O_RDWR`	Open for update (reading and writing).
`w+`	`O_RDWR\|O_CREAT\|O_TRUNC`	Truncate or create for update.
`a+`	`O_RDWR\|O_CREAT\|O_APPEND`	Append: open or create for update at end-of-file.

Binary read and write modes can also be used. These can be specified by using the character `b` in combination with any of the mode parameters shown above. For example, to specify binary read/write mode, use "`rb+`" or "`r+b`". There is no difference in behavior between text streams and binary streams.

The `freopen` function substitutes the named file for the open stream. The original stream is closed, regardless of whether the open ultimately succeeds. The `freopen` function returns a pointer to the `FILE` structure associated with `stream`. The `freopen` function is typically used to attach the previously opened streams associated with `stdin`, `stdout`, and `stderr` to other files.

The `fdopen` function associates a stream with a file descriptor. Thus, `fdopen` can be used to access the file descriptors returned by `open`, `creat`, `dup`, or `fcntl`. (These calls return file descriptors, not pointers to a `FILE` structure.) The type of stream must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening `fseek` or `rewind`, and input may not be directly followed by output without an intervening `fseek`, `rewind`, or an input operation that encounters end-of-file.

When a file is opened for append (that is, when type is `a` or `a+`), it is impossible to overwrite information already in the file. The function `fseek` may be used to reposition the file pointer to any position in the file, but when output is written to the file the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output.

Return values

On success, functions `fopen`, `freopen`, and `fdopen` return a valid file pointer. On failure, NULL is returned.

The maximum number of open `FILE` streams is 20.

Note

The parameter `mode` must have one of the values in the first column in the table; do not use values intended for open, such as `O_RDONLY`.

See also

`fclose, ferror, fread, fseek, getc, gets, open, putc, puts, setbuf, stdio`

File buffering—`setbuf, setvbuf`

Synopsis

```
#include <StdIO.h>
void setbuf (FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Description

A buffer is normally allocated by the Standard C Library at the time of the first `getc` or `putc` on a file. If you prefer to provide your own buffer, you can call `setbuf` or `setvbuf` after a stream has been associated with an open file but before it is read or written. The functions `setbuf` and `setvbuf` let you provide your own buffering for a file stream. The function `setvbuf` is a more flexible extension of `setbuf`.

The `setbuf` function causes the character array pointed to by `buf` to be used instead of an automatically allocated buffer. `BUFSIZ`, a constant defined in the <StdIO.h> header file, lets you specify the size of the `buf` array as

```
char buf[BUFSIZ];
```

If `buf` is `NULL`, input/output is unbuffered.

The `setvbuf` function lets you specify two parameters in addition to those required by `setbuf`: `size` and `mode`. Parameter `size` specifies the size in bytes of the array to be used; the standard I/O functions work most efficiently when `size` is a multiple of `BUFSIZ`. If buffer pointer `buf` is `NULL`, a buffer of `size` bytes is allocated from the system. If `size` is not 0, `size` is assigned to the `FILE` variable's size parameter; if `buf` is not `NULL`, `buf` is assigned to the `FILE` variable's buffer-pointer parameter. The value of `mode` determines how `stream` is buffered by `setvbuf`:

Value of mode	Description
`_IOFBF`	Causes input/output to be file buffered.
`_IOLBF`	Causes output to be line buffered. The buffer is flushed when a newline character is written or when the buffer is full.
`_IONBF`	Causes input/output to be unbuffered. Parameters `buf` and `size` are ignored.

The following function calls are equivalent when `buf` is not `NULL`:

```
setbuf(stream, buf);
setvbuf(stream, buf, _IOFBF, BUFSIZ);
```

The following function calls are equivalent when `buf` is `NULL`:

```
setbuf(stream, NULL);
setvbuf(stream, NULL, _IONBF, 0);
```

Diagnostics

The function `setvbuf` returns nonzero if an invalid value is given for `mode`.

Note

The buffer must have a lifetime at least as great as the open stream. Be sure to close the stream before the buffer is deallocated. If you allocate buffer space as an automatic variable in a code block, be sure to close the stream in the same block.

If `buf` is `NULL` and the system cannot allocate `size` bytes, a smaller buffer will be allocated.

See also

`fclose, fopen, getc, malloc, putc, stdio, ungetc`

Formatted input/output functions

The Standard I/O package provides several functions for formatted input and output. The `printf` function and its relatives send out formatted output to various destinations. The `scanf` function and its relatives read formatted input from various sources.

Printing formatted output—`printf, fprintf, sprintf, vprintf, vfprintf, vsprintf`

Synopsis

```
#include <StdIO.h>
int printf(const  char *format, ... );
int fprintf(FILE *stream, const  char *format, ... );
int sprintf( char *s, const  char *format, ... );

int vprintf(const  char *format, va_list arg);
int vfprintf(FILE *stream, const  char *format, va_list arg);
int vsprintf( char *s, const  char *format, va_list arg);
```

Description

The `printf` function places formatted output on the standard output stream `stdout`. The `fprintf` function places formatted output on the named output stream `stream`. The `sprintf` function places formatted output, followed by the null character (`\0`), into the character array pointed to by `s`; it's your responsibility to ensure that enough room is available. Each function returns the number of characters transmitted (not including the `\0` in the case of `sprintf`), or a negative value if an output error was encountered.

The `vprintf`, `vfprintf`, and `vsprintf` functions handle formatted output in the same manner as `printf`, `fprintf`, and `sprintf`, but the variable argument lists are replaced by `va_list arg`. The `va_list` is initialized by the `va_start` macro, as described later in this chapter under "Variable Arguments—StdArg.h." Each of these functions invokes the `va_arg` macro, but not the `va_end` macro.

Each of these functions converts, formats, and prints its parameters under control of the `format` parameter. The format is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching zero or more parameters. The behavior of the function is undefined if there are insufficient parameters for the format. If the format is exhausted while parameters remain, the extra parameters are ignored.

Each conversion specification is introduced by the character `%`. After `%`, the following elements appear in sequence:

1. Zero or more flag characters, which modify the meaning of the conversion specification.
2. An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded to the field width on the left (default) or right (if the left-adjustment flag has been given); flag specification is given after this list.
3. A precision that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, or `X` conversions; the number of digits to appear after the decimal point for the `e`, `E`, and `f` conversions; the maximum number of significant digits for the `g` and `G` conversions; or the maximum number of characters to be printed from a string in the `s` conversion. The format of the precision is a period (`.`) followed by a decimal digit string; a null digit string is treated as zero.
4. An optional `h` specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion is an `unsigned` or `short int`. An optional `l` specifying that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion character applies to an `arg` parameter of type `long int`. The `l` option is ignored in this implementation because type `int` and type `long` both require 32 bits. An optional `L` specifies that the following `e`, `E`, `f`, `g`, or `G` conversion is a `long double`.
5. A character that indicates the type of conversion to be applied.

◆ *Note:* A field width or precision may be indicated by an asterisk (`*`) instead of a digit string. In this case, an integer `arg` parameter supplies the field width or precision. The `arg` parameter that is actually converted is not fetched until the conversion letter is seen; therefore, the `arg` parameters specifying field width or precision must appear immediately before the `arg` parameter (if any) to be converted.

These are the flag characters and their meanings:

–	The result of the conversion will be left justified within the field.
+	The result of a signed conversion always begins with a sign (+ or –).
space	If the first character of a signed conversion is not a sign, a space will be prefixed to the result. This implies that if the *space* and + flags both appear, the *space* flag will be ignored.
#	The value is to be converted to an alternate form. For `c`, `d`, `s`, and `u` conversions, the flag has no effect. For `o` conversion, it increases the precision to force the first digit of the result to be a zero. For `x` (`X`) conversion, a nonzero result will have `0x` (`0X`) prefixed to it. For `e`, `E`, `f`, `g`, and `G` conversions, the result will always contain a decimal point, even if no digits follow the point. (Normally, a decimal point appears in the result of these conversions only if a digit follows it.) For `g` and `G` conversions, trailing zeros in the fractional part will not be removed from the result (as they normally are).
`0`	The `0` flag pads the field with zeros on the left only; this applies to `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, and `G` conversions. The leading zeros (following any indication of sign or base) pad the field width, and no space padding is performed. If both the `0` and – flags appear, the `0` flag is ignored. If a precision is specified for `d`, `i`, `o`, `u`, `x`, and `X` conversions, the `0` flag is ignored. For other conversion, the behavior is not defined.

These are the conversion characters and their meanings:

`d`, `i`, `o`, `u`, `x`, `X`	The integer `arg` parameter is converted to signed decimal (`d` or `i`), unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` or `X`), respectively; the letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.
`n`	This specifies a pointer to an integer into which the function will store the number of characters read so far.

`f` The `float`, `double`, `comp`, or `extended` `arg` parameter is converted to decimal notation in the form "[–]*ddd*.*ddd*", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, it is assumed to be 6; if the precision is explicitly 0, no decimal point appears. Infinities are printed in the form "[–]INF", and NaNs are printed in the form "[–]NAN (*ddd*)", where *ddd* is a code indicating why the result is not a number.

`e, E` The `float`, `double`, `comp`, or `extended` `arg` parameter is converted in the form "[–]`d`.*ddd* `e`±*dd*", where there is one digit before the decimal point, and where the number of digits after the decimal point is equal to the precision. When the precision is missing, it is assumed to be 6; if the precision is 0, no decimal point appears. The `E` format code produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits. Infinities are printed as INF, and NaNs are printed in the form "[–]NAN (*ddd*)", where *ddd* is a code indicating why the result is not a number.

`g, G` The `float`, `double`, `comp`, or `extended` `arg` parameter is printed in style `f` or `e` (or in style `f` or `E` in the case of a `G` format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style `e` is used only if the exponent resulting from the conversion is less than –4 or greater than the precision. Trailing zeros are removed from the result. A decimal point appears only if it is followed by a digit.

`c` The character `arg` parameter is printed.

`s` The `arg` parameter is taken to be a string (character pointer) and characters from the string are printed until a null character (`\0`) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. If the string pointer `arg` parameter has the value zero, the result is undefined; a zero `arg` parameter yields undefined results.

`p` The `arg` parameter is taken to be a pointer.

`P` The `arg` parameter is taken to be a Pascal string, which begins with a character specifying its length and does not end with a null character (\0). (This conversion character was `%p` in MPW 2.0 C).

`%` Print a `%`; no parameter is converted.

In no case does a nonexistent or small field width cause truncation of a field. If the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by `printf` and `fprintf` are printed as if `putc` had been called.

Examples

To print a date and time in the form "Sunday, July 3, 10:02", where `weekday` and `month` are pointers to null-terminated strings:

```
printf("%s, %s %d, %.2d:%.2d", weekday, month, day, hour, min);
```

To print pi to five decimal places:

```
printf("pi = %.5f", pi());
```

Note

Calling `sprintf` causes other Standard I/O functions to be loaded, even though `sprintf` doesn't perform any I/O.

See also

`ecvt, fread, putc, puts, scanf, stdio`

Converting formatted input—`scanf, fscanf, sscanf`

Synopsis

```
#include <StdIO.h>
int scanf( const  char *format, ... );
int fscanf(FILE *stream, const  char *format, ... );
int sscanf(const  char *s, const  char *format, ... );
```

Description

The `scanf` function reads characters from the standard input stream, `stdin`. The `fscanf` function reads characters from the named input stream `stream`. The `sscanf` function reads characters from the character string `s`. Each function converts the input according to a control string (`format`), then stores each converted value into the memory pointing to the corresponding variable parameter.

The `format` parameter, the control string, contains specifications that control the interpretation of input sequences. The format consists of characters to be matched in the input stream and/or conversion specifications that start with the character `%`. The control string may contain

- White-space characters (spaces and tabs) that cause input to be read up to the next non-white-space character, except as described later in this section.
- A character (any except `%`) that must match the next character of the input stream. (To match a `%` character in the input stream, use `%%`.)

- Conversion specifications beginning with the character `%` and followed by an optional assignment suppression character `*`, an optional numeric maximum field width, an optional `l`, `L`, or `h` indicating the size of the receiving parameter, and a conversion code.

An input field is defined relative to its conversion specification. The input field ends when the first character inappropriate for conversion is encountered or when the specified field width is exhausted. After conversion, the input pointer points to the inappropriate character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding parameter, which is a pointer to a basic C type such as `int` or `float`.

Assignment can be suppressed by preceding a format character with the character `*`. *Assignment suppression* means an input field is skipped: the field is read and converted but not assigned. Therefore, a corresponding pointer argument should be omitted for each suppressed input field.

The *format character* dictates the interpretation of the input field. The following format characters are legal in a conversion specification, after `%`:

`%` A single `%` is expected in the input at this point; no assignment is done.

`d` A decimal integer is expected; the corresponding parameter should be an integer pointer.

`i` A pointer to an integer is expected as the argument, and the input field is interpreted as an integer. The format of the number determines its base. If the first characters are `0x` or `0X`, the number is interpreted as hexadecimal. If the first character is `0`, the number is interpreted as octal. In all other cases, the number is interpreted as decimal.

`n` This specifies a pointer to an integer into which the function will store the number of characters read so far.

`u` An unsigned decimal integer is expected; the corresponding parameter should be an unsigned integer pointer.

`o` An octal integer is expected; the corresponding parameter should be an integer pointer.

`x` A hexadecimal integer is expected; the corresponding parameter should be an integer pointer.

The conversion characters `d`, `u`, `o`, and `x` may be preceded by `l` or `h` to indicate that a pointer to `long` or `short`, rather than `int`, is in the parameter list. The `l` is ignored in this implementation because `int` and `long` are both 32 bits.

`e, f, g` A floating-point number is expected; the next field is converted accordingly and stored through the corresponding parameter, which should be a pointer to a `float`, `double`, `comp`, or `extended`, depending on the size specification. The input format for floating-point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of `E` or `e` followed by an optionally signed integer. In addition, infinity is represented by the string `"INF"`, and NaNs are represented by the string `"NAN"`, optionally followed by parentheses that may contain a string of digits (the NaN code). Case is ignored in the infinity and NaN strings.

The conversion characters `e`, `f`, and `g` may be preceded by `l`, `m`, or `L` to indicate that a pointer to `double`, `comp`, or `extended`, rather than `float`, is in the parameter list.

`s` A character string is expected; the corresponding parameter should be a character pointer to an array of characters large enough to accept the string; a terminating null character (`\0`) is added automatically. The input field is terminated by a white-space (blank or tab) character, or when the number of characters specified by the maximum field width has been read.

`p` A pointer is expected as the argument.

`P` A character string is expected. The next field is converted to a Pascal string, that is, a length byte followed by the string itself, without a terminating null character (`\0`). The corresponding parameter should be a pointer to an array of characters large enough to accept the string. The input field is terminated by a white space (blank or tab) character, or when the number of characters specified by the maximum field width has been read. (This was `%p` in MPW 2.0 C).

`c` A character is expected; the corresponding parameter should be a character pointer. The normal skip over white space is suppressed in this case; use `%1s` (digit one) to read the next non-white-space character. If a field width is given, the corresponding parameter should refer to a character array; the indicated number of characters is read.

`[` The left bracket introduces a *scanset* format. The input field is the maximal sequence of input characters consisting entirely of characters in the scanset. When reading the input field, the normal skip over leading white space is suppressed. The corresponding pointer parameter must point to a character array large enough to hold the input field and the terminating null character (`\0`), which will be added automatically. The left bracket is followed by a set of characters (the scanset) and a terminating right bracket.

^ When it appears as the first character in the scanset, the circumflex (^) serves as a complement operator and redefines the scanset as the set of all characters not contained in the remainder of the scanset string.

] The right bracket ends the scanset. To be included as an element of the scanset, the right bracket must appear as the first character (possibly preceded by a circumflex) of the scanset. Otherwise, it will be interpreted syntactically as the closing bracket.

A range of characters may be represented by the construct *first–last;* thus the scanset `[0123456789]` may be expressed `[0-9]`. To use this convention, *first* must be less than or equal to *last* in the ASCII collating sequence. Otherwise, the minus (–) will stand for itself in the scanset. The minus will also stand for itself whenever it is the first or the last character in the scanset.

At least one character must match for the conversion to be considered successful.

Conversion terminates at `EOF`, at the end of the control string, or when an input character doesn't match the control string. In the last case, the unmatched character is left unread in the input stream.

Examples

The following examples show how `scanf` works.

Example 1:

The call

```
int i;
float x;
char name[50];
scanf("%d%f%s", &i, &x, name);
```

with input

```
25 54.32E-1 Louisa
```

will assign the value 25 to `i` and value 5.432 to `x`, and put `Louisa\0` into `name`.

Example 2:

The call

```
int i;
extended x;
char name[50];
scanf("%2d%nf%*d %[0-9]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign the value 56 to `i` and the value 789.0 to `x`, skip `0123`, and put `56\0` into `name`. The next call to `getchar` will return `a`.

Example 3:

The call

```
int i;
scanf("answer1=%d", &i);
```

with input

```
answer1=51 answer2=45
```

will assign the value 51 to `i` because `"answer1"` is matched explicitly in the input stream; the input pointer will be left at the space before `"answer2"`.

Return value

The `scanf`, `fscanf`, and `sscanf` functions return the number of successfully matched and assigned input items; this number can be 0 when an early mismatch between an input character and the control string occurs. If the input ends before the first mismatch or conversion, `EOF` is returned.

These functions return `EOF` on end of input and a short count for missing or illegal data items.

Note

Trailing white space is left unread unless matched in the control string.

The success of literal matches and suppressed assignments is not directly determinable.

Warning

The pointer variable parameters to `scanf`, `fscanf`, and `sscanf` must be pointers—for example, `&i`. Be sure to pass the address of `i` rather than its value.

See also

`atof, atoi, fread, getc, gets, printf, stdio, strtol`

Apple Numerics Manual, 2nd edition

Character input/output functions

The Standard I/O package contains a number of functions for getting character and string input and sending character and string output.

Getting character input—`getc`, `getchar`, `fgetc`, `getw`

Synopsis

```
#include <StdIO.h>
int getc (FILE *stream);
int getchar(void);
int fgetc (FILE *stream);
int getw (FILE *stream);
```

Description

The `getc` macro returns the next character from the named input stream. It also moves the file pointer, if defined, ahead one character in `stream`. The macro `getc` cannot be used if a function is necessary; for example, you cannot have a function pointer point to it. The macro `getc` returns the integer `EOF` on end-of-file or error.

The `getchar` macro returns the next character from the standard input stream, `stdin`.

The `fgetc` function produces the same result as macro `getc`; function `fgetc` runs more slowly than macro `getc` but takes less space per invocation. You can also have a pointer to `fgetc` but not to `getc`.

The `getw` function returns the next `int` (that is, four bytes) from the named input stream so that the order of bytes in the stream corresponds to the order of bytes in memory. The function `getw` returns the constant `EOF` upon end-of-file or error. Because `EOF` is a valid integer value, `feof` and `ferror` should be used to check the success of `getw`. The function `getw` increments the associated file pointer, if defined, to point to the next `A`. The function `A` assumes no special alignment in the file.

Return values

These calls return data from the stream, or the integer constant `EOF` (–1) at end-of-file or upon an error.

Note

Because it is implemented as a macro, `getc` treats a stream parameter with side effects incorrectly. In particular,

```
getc(*f++)
```

doesn't work as you would expect. Instead use

```
fgetc(*f++)
```

See also

`ferror, fopen, fread, gets, putc, scanf, setbuf, stdio, ungetc`

Getting string input—`gets, fgets`

Synopsis

```
#include <StdIO.h>
char *gets( char *s)
char *fgets( char *s, int n, FILE *stream);
```

Description

The `gets` function reads characters from the standard input stream, `stdin`, into the array pointed to by `s` until a newline character is read or an end-of-file condition is encountered. The newline character is discarded, and the string is terminated with a null (`\0`) character. (For more information about newline, see "Character Constants: Newline, Carriage=Return, and Vertical=Tab" in Chapter 2.)

The `fgets` function reads characters from `stream` into the array pointed to by `s` until `maxlen-1` characters are read, a newline character is read and transferred to `s`, or an end-of-file condition is encountered. The string is then terminated with a null character.

Return values

If end-of-file is encountered and no characters have been read, no characters are transferred to `s` and `NULL` is returned. If a read error occurs, `NULL` is returned. Otherwise `s` is returned. (A read error will occur, for example, if you attempt to use these functions on a file that has not been opened for reading.)

Note

The array pointed to by `s` is assumed to be large enough; overflow is not checked.

The function `gets` omits the newline character in the string; `fgets` leaves it in.

See also

`ferror, fopen, fread, getc, scanf, stdio`

Character output—`putc`, `putchar`, `fputc`, `putw`

Synopsis

```
#include <StdIO.h>
int putc(int c, FILE *stream);
int putchar(int c);
int fputc(int c, FILE *stream);
int putw(int w, FILE *stream);
```

Description

The `putc` macro writes the integer `c` to the output stream at the current position of the file pointer.

The macro call `putchar(c)` is equivalent to `putc(c, stdout)`.

The `fputc` function behaves like the `putc` macro. The function `fputc` runs more slowly than macro `putc` but takes less space per invocation.

The `putw` function writes an `int` (that is, four bytes) to the output stream at the current position of the file pointer. This function neither assumes nor causes special alignment in the file.

For information about buffering of output files, see the "Standard I/O Buffering" section, earlier in this chapter.

Return values

On success, the `putc`, `putchar`, and `fputc` functions return zero. On failure, they return the constant `EOF`. This occurs if the file stream is not open for writing or if the output file cannot be grown. The `putw` function returns zero on success and a nonzero value on error.

Note

Because it is implemented as a macro, `putc` treats a stream parameter with side effects incorrectly. In particular, `putc(c, *f++)` produces unexpected results. Instead, use `fputc(c, *f++)`.

See also

`fclose, ferror, fopen, fread, getc, printf, puts, setbuf, stdio`

String output—puts, fputs

Synopsis

```
#include <StdIO.h>
int puts(const char *s);
int fputs(const char *s, FILE *stream);
```

Description

The `puts` function writes the null-terminated string pointed to by `s`, followed by a newline character, to the standard output stream `stdout`.

The `fputs` function writes the null-terminated string pointed to by `s` to the named output stream `stream`.

Neither function writes the terminating null character.

Return value

Both routines return the number of characters written, or an `EOF` if there is a write error.

Note

The `puts` function appends a newline character, whereas `fputs` does not.

See also

`ferror, fopen, fread, printf, putc, stdio`

Undoing an input—ungetc

Synopsis

```
#include <StdIO.h>
int ungetc (int c, FILE *stream);
```

Description

The `ungetc` function inserts the integer `c` into the buffer associated with an input stream. The stream must be file buffered or line buffered; it cannot be unbuffered. The inserted character, `c`, will be returned by the next `getc` call on that stream. The `ungetc` function returns `c` and leaves the file stream unchanged.

Only one character of pushback is allowed, provided something has been read from the stream and the stream is not unbuffered.

If `c` equals `EOF`, `ungetc` does nothing to the buffer and returns `EOF`. In other words, you cannot use `ungetc` to force yourself to the end-of-file the next time you read the file.

The `fseek` function undoes the effect of `ungetc`.

Diagnostics

For `ungetc` to perform correctly, a read must have been performed before the call to the `ungetc` function. The function `ungetc` returns `EOF` if it can't insert the character.

Note

The function `ungetc` does not work on unbuffered streams.

See also

`fseek, getc, setbuf, stdio`

Direct input/output—`fread, fwrite`

Synopsis

```
#include <StdIO.h>
size_t fread( void *ptr, size_t size, size_t nmemb, FILE *stream);
int fwrite(const  void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Description

The `fread` function copies `nmemb` items of data from the named input stream into an array beginning at `ptr`. An item of data is a sequence of `size` bytes (not necessarily terminated by a null byte). The function `fread` stops appending bytes if an end-of-file or error condition is encountered while reading stream or if `nmemb` items have been read. The function `fread` leaves the file pointer in `stream` pointing to the byte following the last byte read.

The `fwrite` function writes at most `nmemb` items of data to the named output stream from the array pointed to by `ptr`. An item is a sequence of `size` bytes. The `fwrite` function stops writing when it has written `nmemb` items of data or if an error condition is encountered on `stream`. The `fwrite` function doesn't change the contents of the array pointed to by `ptr`.

The `size` parameter is typically `sizeof(*ptr)`, where `sizeof` specifies the length of an item pointed to by `ptr`.

Return values

The `fread` and `fwrite` functions return the number of items read or written. If `nmemb` is 0 or negative, no characters are read or written and 0 is returned by both `fread` and `fwrite`.

See also

`fopen, getc, gets, printf, putc, puts, read, scanf, stdio, write`

File positioning—`fseek, rewind, ftell, fgetpos, fsetpos`

Synopsis

```
#include <StdIO.h>
#define SEEK_SET  0
#define SEEK_CUR  1
#define SEEK_END  2

int fseek(FILE *stream, long int offset, int whence);
void rewind(FILE *stream);
long int ftell(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description

The `fseek` function sets the position of the next input or output operation on the stream.

The value of `whence` determines how the `offset` parameter is used, according to these rules:

- If `whence` is `SEEK_SET`, the new position is `offset` bytes from the beginning of the file.
- If `whence` is `SEEK_CUR`, the new position is the current location plus `offset`.
- If `whence` is `SEEK_END`, the new position is the size of the file plus `offset`.
- If `whence` is `SEEK_SET` or `SEEK_CUR`, the value of `offset` may be negative.

The call `rewind(stream)` is equivalent to `fseek(stream, 0L, 0)` except that the `rewind` call returns no value.

The `fseek` and `rewind` functions undo any effects of `ungetc`.

After `fseek` or `rewind`, the next operation on a file opened for update may be either input or output.

The `ftell` function returns the offset of the current byte relative to the beginning of the file associated with the named stream.

The `fgetpos` function stores the current value of the file position indicator for `stream` in the object that `pos` points to. This value contains information used by `fsetpos`, which repositions the stream to the position it held at the time the `fgetpos` function was called.

The `fsetpos` function is used in conjunction with the the `fgetpos` function. The `fsetpos` function determines the position for `stream` according to the value of `pos` returned by an earlier call to the `fgetpos` function on the same stream.

If a call to the `fsetpos` function is successful, it clears the end-of-file indicator for the stream, undoing any effects of the `ungetc` function on that stream. After an `fsetpos` call on the stream, the next operation may be either input or output.

Return values

The `fseek` function returns nonzero for improper seeks; otherwise it returns zero. An example of an improper seek is an `fseek` before the beginning of, or past the end of, the file.

The `getpos` and `fsetpos` functions return zero if successful. If the `fgetpos` or `fsetpos` functions fail, they return nonzero and place a positive value in `errno`.

See also

`lseek, fopen, stdio, ungetc`

Error-handling—`feof, ferror, clearerr, perror, fileno`

Synopsis

```
#include <StdIO.h>
int feof(FILE *stream);
int ferror(FILE *stream);
void clearerr(FILE *stream);
void perror (const char *s);
int fileno(FILE *stream);
```

Description

The `feof` macro returns nonzero when the end-of-file has previously been detected while reading the named input stream; otherwise, it returns zero.

The `ferror` macro returns nonzero when an I/O error has previously occurred while reading from or writing to the named stream; otherwise, it returns zero.

The `clearerr` macro resets the error indicator and end-of-file indicator to zero on the named stream.

The `perror` function takes a string as its argument. If the string is not null, `perror` prints the string, plus a colon and a space. In any case, it then prints an appropriate error message derived from the current value of `errno`. (This message is the same one that `strerror` would print.)

The `fileno` macro returns the integer file descriptor associated with the named stream, as explained in the section on the `open` function.

See also

`close, fcntl, fopen, getc, gets, open, putc, puts, stdio`

Input/Output control (non-ANSI extension)—IOCtl.h

The IOCtl.h header file contains a single function, `ioctl`, which is used for communication with device drivers.

The `ioctl` function

Synopsis

```
#include <IOCtl.h>
int ioctl(int fildes, unsigned int cmd, long *arg);
```

Description

The `ioctl` function communicates with a file's device driver by sending control information, requesting status information, or both. Parameter `cmd` indicates which device-specific operations `ioctl` must perform. Here are the control values.

Value of cmd	Description
`FIOINTERACTIVE`	The `ioctl` function returns 0 if the device is interactive; if not, it returns –1 and `errno` is set to `EINVAL`. Parameter `arg` is ignored.
`FIOBUFSIZE`	The `ioctl` function returns, in bytes, the optimal buffer size for this device; the buffer size is returned in a `long` pointed to by `arg`. If the device has no default buffer size, `ioctl` returns –1 and `errno` is set to `EINVAL`.
`FIOFNAME`	The `ioctl` function stores the filename associated with `fildes` in a C string pointed to by `arg`. It returns –1 if the filename exceeds 255 characters [`E2BIG`]. Use only for a program running as an MPW tool.
`FIOREFNUM`	The `ioctl` function returns the Macintosh file reference number associated with `fildes`; the reference number is returned in the `short` pointed to by `arg`. If the `fildes` is not open on a Macintosh file (such as the console device), `ioctl` returns –1.
`FIOSETEOF`	The `ioctl` function sets the logical end-of-file specified in the `long` parameter `arg`. The value of `arg` is the new size of the file, in bytes. This command can be used to reduce or increase the size of the open file. The current file pointer is not affected unless the file size is set below it.
`TIOFLUSH`	Used only for the console device and other terminal devices. The `ioctl` function returns –1 if `fildes` is not a terminal device. `TIOFLUSH` tells the device driver to throw away unread terminal input. Parameter `arg` is ignored.

Diagnostics

If an error has occurred, a value of –1 is returned and `errno` is set to indicate the error.

Note

For `cmd` values `FIOINTERACTIVE` and `FIOBUFSIZE`, a function return of –1 is a meaningful response, not an error. For `FIOINTERACTIVE`, `errno` is set to `EINVAL` for devices that are not interactive. For `FIOBUFSIZE`, `errno` is set to `EINVAL` for devices that have no default buffering.

The `cmd` values `FIOLSEEK` and `FIODUPFD` are reserved for operating-system use.

If you set the console GrafPort with `TIOSPORT`, do not deallocate the storage for that port; the console device is written to by the `exit` function as your application terminates.

The console device `ioctl` control values `TIOGPORT` and `TIOSPORT` are no longer supported. It is no longer possible to perform direct I/O to a user-supplied GrafPort.

Warning

`FIOREFNUM` lets you do Macintosh I/O operations, such as `Allocate`, that are not available through `ioctl`. Do not close or modify the file pointer by using the reference number.

See also

`fcntl, faccess, ferror`

Low-level file I/O (non-ANSI extensions)—FCntl.h

MPW 3.0 C contains several file-control functions in addition to those required by the ANSI draft C standard. These functions, declared in the header file FCntl.h, preserve compatibility with MPW 2.0 C.

Some of these functions duplicate, at a lower level, functions in the header file StdIO.h. (For example, `open` is a low-level equivalent of `fopen`.) Using the StdIO.h functions whenever possible will improve portability and robustness.

Creating a file—`creat`

Synopsis

```
#include <FCntl.h>
int creat(const char *filename);
```

Description

The `creat` function creates a new file or prepares to rewrite an existing file, `filename`. If the file exists, the length of its data fork is set to 0.

The function `creat(filename)` is equivalent to

```
open(filename, O_WRONLY|O_TRUNC|O_CREAT)
```

Upon successful completion, a nonnegative integer (the file descriptor) is returned and the file is open for writing. The file pointer is set to the beginning of the file. A maximum of about 30 files may be open at a given time: the actual maximum depends upon the current system environment.

Return value

Upon successful completion, a nonnegative integer (the file descriptor) is returned. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

Note

Other implementations of `creat` specify a second parameter, `mode`. This version ignores any second parameter.

See also

`close, open, read, write`

Opening a file—`open`

Synopsis

```
#include <FCntl.h>
int open(const char *filename, int oflag);
```

Description

The `open` function opens a file descriptor for the named file and sets the file-status flags according to the value of `oflag`. The parameter `filename` is a disk file, window, selection, or pseudofile. (See "Pseudo-Filenames" in Chapter 5 of the *Macintosh Programmer's Workshop 3.0 Reference* for more information.) The value of `oflag` is constructed by performing an OR operation on the flag settings, for example:

```
fildes = open("MyFile", O_WRONLY|O_CREAT|O_TRUNC);
```

To construct `oflag`, first select one of the following access modes:

`O_RDONLY`	Open for reading only.
`WR_ONLY`	Open for writing only.
`O_RDWR`	Open for reading and writing.

Then optionally add one or more of these modifiers:

`O_APPEND`	The file pointer is set to the end of the file before each write.
`O_CREAT`	If the file does not exist, it is created.
`O_TRUNC`	If the file exists, its length is truncated to 0; the mode is unchanged.
`O_RSRC`	The file's resource fork is opened. (Normally, the data fork is opened.)
`O_BINARY`	Open as a binary stream (supported but not used by the system).

The following setting is valid only if `O_CREAT` is also specified:

`O_EXCL`	The function `open` fails if the file exists.

Upon successful completion, a nonnegative integer (the file descriptor) is returned. The file pointer used to mark the current position within the file is set to the beginning of the file.

The named file is opened unless one or more of the following error conditions is true:

`O_CREAT` is not set and the named file does not exist [`ENOENT`].

More than about 30 file descriptors are currently open. The actual limit varies according to run-time conditions [`EMFILE`].

`O_CREAT` and `O_EXCL` are set and the named file exists [`EEXIST`].

Return value

Upon successful completion, a nonnegative integer (the file descriptor) is returned. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

See also

close, creat, dup, fcntl, ferror, fopen, lseek, read, stdio, write

Closing a file—close

Synopsis

```
#include <FCntl.h>
int close(int fildes);
```

Description

The close function closes the file descriptor indicated by fildes. Parameter fildes is a file descriptor obtained from an open, creat, dup, or fcntl call.

The function close fails if fildes is not a valid open file descriptor [EBADF].

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and errno is set to indicate the error.

See also

creat, dup, fclose, fcntl, ferror, open, stdio

Removing a file—unlink

Synopsis

```
#include <FCntl.h>
int unlink(const char *fileName);
```

Description

The unlink function deletes the named file. The function fails if the named file is open. This function is the UNIX (and MPW 2.0 C) equivalent of the ANSI remove function, and is included for compatibility.

A call to `unlink` is equivalent to

```
faccess(fileName, F_DELETE, 0)
```

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

See also

`faccess`

Reading from a file—`read`

Synopsis

```
#include <Fcntl.h>
int read(int fildes, char* buf, unsigned long nbyte);
```

Description

The `read` function transfers up to `nbyte` bytes from the file associated with `fildes` into the buffer pointed to by `buf`. File descriptor `fildes` is obtained from a call to `open`, `creat`, `a`, or `a`.

On devices capable of seeking, `read` starts reading at the current position of the file pointer associated with `fildes`. Upon return from `read`, the file pointer is incremented by the number of bytes actually read.

Nonseeking devices always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, `read` returns the number of bytes actually read and placed in the buffer. This number may be less than `nbyte` if the file is associated with a window or if the number of bytes left in the file is less than `nbyte` bytes. A value of 0 is returned when an end-of-file has been reached, and a value of –1 is returned if a read error occurred.

The function `read` fails if `fildes` is not a valid file descriptor open for reading. [`EBADF`]

File descriptor 0 is opened by the MPW Shell as standard input.

Return values

Upon successful completion, a nonnegative integer is returned, indicating the number of bytes actually read. Otherwise, –1 is returned and `errno` is set to indicate the error.

See also

`creat, ferror, fread, open, stdio`

Writing to a file—`write`

Synopsis

```
#include <Fcntl.h>
int write(int fildes, const char* buf, unsigned long nbyte);
```

Description

The `write` function attempts to write `nbyte` bytes from the buffer pointed to by `buf` to the file associated with the `fildes`. File descriptor `fildes` is obtained from an `open`, `creat`, `dup`, or `fcntl` call. Internal limitations may cause `write` to write fewer bytes than requested; the number of bytes actually written is indicated by the return value. Several calls to `write` may therefore be necessary to write out the contents of `buf`.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from `write`, the file pointer is incremented by the number of bytes actually written.

On nonseeking devices, writing starts at the current position. The value of a file pointer associated with such a device is undefined.

If the `O_APPEND` file-status flag set in `open` is on, the file pointer is set to end-of-file before each write.

The file pointer remains unchanged and `write` fails if `fildes` is not a valid file descriptor open for writing [`EBADF`].

If you try to write more bytes than there is room for on the device, `write` writes as many bytes as possible. For example, if `nbyte` is 512 and there is room for 20 bytes more on the device, `write` writes 20 bytes and returns a value of 20. The next attempt to write a nonzero number of bytes will return an error [`ENOSPC`].

File descriptor 1 is opened by the MPW Shell as standard output; file descriptor 2, as standard error.

Return value

Upon successful completion, the number of bytes actually written is returned. Otherwise, –1 is returned and `errno` is set to indicate the error.

See also

`creat, ferror, fread, lseek, open, stdio`

File positioning—`lseek`

Synopsis

```
#include <FCntl.h>
#define SEEK_SET  0
#define SEEK_CUR  1
#define SEEK_END  2

long lseek(int fildes, long int offset, int whence);
```

Description

A file descriptor, `fildes`, is returned from a call to `creat`, `dup`, `fcntl`, or `open`. The `lseek` function sets the file pointer associated with `fildes`, thus determining the position of the next read or write operation.

The value of `whence` determines how the `offset` parameter is used, according to these rules:

- If `whence` is `SEEK_SET`, the new position is `offset` bytes from the beginning of the file.
- If `whence` is `SEEK_CUR`, the new position is the current location plus `offset`.
- If `whence` is `SEEK_END`, the new position is the size of the file plus `offset`.
- If `whence` is `SEEK_SET` or `SEEK_CUR`, the value of `offset` may be negative.

Upon successful completion, the file-pointer value as measured in bytes from the beginning of the file is returned.

The file pointer remains unchanged and `lseek` fails if one or more of the following error conditions are true:

- File descriptor `fildes` is not open [EBADF].
- Parameter `whence` is not equal to SEEK_SET, SEEK_CUR, or SEEK_END [EINVAL].
- The resulting file pointer would point past the end-of-file [ESPIPE].
- The resulting file pointer would point before the beginning of the file [EINVAL].

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

Return values

Upon successful completion, a nonnegative long integer indicating the file-pointer value is returned. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

Note

In previous versions of the Standard C Library, `tell(fildes)` was a function that returned the current file position. It is equivalent to the call `lseek(fildes, 0L, 1)`.

Warning

The `lseek` function has no effect on a file opened with the O_APPEND flag because the next write to the file always repositions the file pointer to the end before writing.

See also

`ferror, fseek, open, stdio, write`

Get file control and status information—`faccess`

Synopsis

```
#include <FCntl.h>
int faccess(const char *filename, unsigned int cmd, long *arg);
```

Description

The `faccess` function provides access to control and status information for named files. (Compare function `ioctl`, which provides different control and status information for open files.)

The `cmd` parameter must be set to one of the constants in the following list to indicate what operation is to be performed on the file. As noted in the list, some calls to `faccess` also require the `arg` parameter, usually as a `long` or as a pointer to a `long`.

The following commands are available to all programs:

Value of cmd	Description
`F_DELETE`	Deletes the named file, or returns an error if the file is open. Parameter `arg` is ignored.
`F_RENAME`	Renames the named file. Parameter `arg` is a pointer to a string containing the new name.

The following commands can be used only by MPW tools:

Value of cmd	Description
`F_GTABINFO`	Gets the tab offset for the MPW text file `filename`. The tab offset is stored in the long integer pointed to by `arg`. The tab offset is expressed as an integer number of spaces. The width of the space character in the current font determines the actual distance of the tab offset.
`F_STABINFO`	Sets the tab offset for the MPW text file `filename`. The tab offset is specified as a `long` value in `arg`.
`F_GFONTINFO`	Gets the font number and font size for an MPW text file `filename`. The font number is stored in the high-order half of the `long` pointed to by `arg`; the font size is stored in the low order half of the same `long`.
`F_SFONTINFO`	Sets the font number and font size for the MPW text file, `filename`. The font number is specified in the high-order half of `arg`; the font size is specified in the low-order half of `arg`.
`F_GPRINTREC`	Gets a print record `TPrint` for an MPW text file, `filename`; `arg` is a handle to the print record.
`F_SPRINTREC`	Sets a print record for the MPW text file `filename`; `arg` is a handle to the print record.
`F_OPEN`	Reserved for operating system use.

(continued)

Value of cmd	Description
F_GSELINFO	Gets the selection information for the MPW text file filename.
F_SSELINFO	Sets the selection information for the MPW text file filename.
F_GWININFO	Gets the current window position.
F_SWININFO	Sets the current window position.

The commands F_GTABINFO and F_GFONTINFO pass arg as a pointer to a long; F_STABINFO and F_SFONTINFO pass arg as a long value; and F_GPRINTREC and F_SPRINTREC pass arg as a handle to a print record.

Return values

Upon successful completion, faccess returns a nonnegative value, usually 0. If the device for the named file cannot perform the requested command, faccess returns –1 and errno is set to indicate the error.

If the requested resource for F_GTABINFO, F_GFONTINFO, or F_GPRINTREC does not exist for the named file, default values are stored and the function returns a value greater than 0.

Note

Before faccess is called with F_GPRINTREC or F_SPRINTREC, the Printing Manager must be initialized and the print record handle THPrint must be allocated. The font size must be 9, 10, 12, 14, 18, or 24; the font number must be 0 or a positive integer. The following sequence must be used with these print command values:

```
res = CurResFile();
PRClose();
UseResFile(res);
PROpen();
   /* do whatever, including call faccess print commands */
PRClose();
UseResFile(res);
```

See also

ioctl, unlink

Duplicate an open file descriptor—dup

Synopsis

```
#include <Fcntl.h>
int dup(int fildes);
```

Description

The dup function returns a new file descriptor with these features:

- It refers to the same open file as the original.
- It uses the same file pointer as the original.
- It has the same access mode (that is, read, write, or read/write) as the original.

The fildes parameter is a file descriptor obtained from an open, creat, dup, or fcntl call. The new file descriptor returned by dup is the lowest one available.

The function call dup(fildes) is equivalent to fcntl(fildes, F_DUPFD, 0).

The dup function fails if parameter fildes is not a valid open file descriptor [EBADF].

Return value

Upon successful completion, a nonnegative integer (the file descriptor) is returned. Otherwise, a value of –1 is returned and errno is set to indicate the error.

See also

close, fcntl, ferror, open

Duplicate a file descriptor—fcntl

Synopsis

```
#include <FCntl.h>
int fcntl(int fildes, unsigned int cmd, int arg);
```

Description

The `fcntl` function duplicates a file descriptor. A file remains open until all of its file descriptors are closed.

The `fildes` parameter is an open file descriptor obtained from an `open`, `creat`, `dup`, or `fcntl` call. The `cmd` parameter takes the value `F_DUPFD`, which tells `fcntl` to return the lowest-numbered available file descriptor greater than or equal to `arg`. Normally `arg` is greater than or equal to 3, to avoid obtaining the standard file descriptors 0, 1, and 2. The function `fcntl` returns a new file descriptor that points to the same open file as `fildes`. The new file descriptor has the same access mode (read, write, or read/write) and file pointer as `fildes`. Any I/O operation changes the file pointer for all file descriptors that share it.

The `fcntl` function fails if one or more of the following error conditions are true:

- Parameter `fildes` is not a valid open file descriptor [`EBADF`].
- Parameter `arg` is negative or greater than the highest allowable file descriptor [`EINVAL`].

Return values

Upon successful completion, the value returned is a new file descriptor. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

Note

MPW 3.0 C does not support the `F_GETFD`, `F_SETFD`, `F_GETFL`, and `F_SETFL` commands of `fcntl`.

See also

`close, dup, ferror, ioctl, open`

Variable arguments—StdArg.h

Synopsis

```
#include <StdArg.h>
typedef char *va_list;  /*character pointer */

#define va_start(va_list ap, parmN)
#define va_arg(va_list ap, type)
#define va_end(va_list ap) /* do nothing */
```

Description

The `va_start` macro initializes a list of arguments for later use by the `va_arg` and `va_end` macros. The `va_start` macro takes two arguments: the variable list `ap` used to refer to the optional arguments, and the name of the last named argument `parmN` passed to the function. It returns no value.

The `va_arg` macro returns a value for each of the optional arguments contained in the `va_list` initialized by `va_start`. Successive uses of `va_arg` return both the current position and the type of each of the arguments in `va_list`.

The `va_end` macro does any necessary cleanup after using optional arguments with `va_arg`. It returns no value, but simply allows a normal return from `va_arg`.

Example

For a sample function declaration taking two or more arguments:

```
#include <StdArg.h>
/* ANSI function declarations use ', ...'to indicate
   the position of the first optional argument.*/
      .
int foo(char *arg1, int arg2, ...);
   {
   va_list nextArg;          /* A local variable, nextArg,
                                indicates the current position in the
                                list of optional arguments */
   va_start(nextArg, arg2);/* set up va_list */
   va_arg(nextArg, int);     /* refer to each optional argument in turn */
      .
      .
      .
   va_end(nextArg);          /* cleanup after va_arg */
   }
```

Since the compiler widens many argument types when they are pushed on the stack, it is important to remember that the argument type in `va_arg` should represent the type whose length corresponds to the length of the argument on the stack. Here are some examples.

Type	Type pushed on stack
`float`	`extended`
`double`	`extended`
`char`	`int`
`short`	`int`

Note

The StdArg.h header file was named VarArg.h in previous versions of MPW C.

See also

Formatted input/output

General utilities—StdLib.h

Synopsis

```
#include <StdLib.h>

#define  NULL 0

typedef struct {     /* structure type returned by div function */
   int quot;
   int rem;
   }  div_t;

typedef struct {     /* structure type returned by ldiv function */
   long int quot;
   long int rem;
   }  ldiv_t;

#define EXIT_FAILURE 1      /* arguments to the exit function */
#define EXIT_SUCCESS 0

#define RAND_MAX 32767      /* maximum that rand function can return */
#define MB_CUR_MAX 1        /* maximum number of bytes in a
                               multibyte character */
```

String-conversion functions

The Standard C Library provides functions to convert strings to integers (`int` and `long`) floating-point numbers (`float` and `double`).

String-to-`int` conversion—`atoi`, `atol`

Synopsis

```
#include <StdLib.h>
int atoi(const char *nptr);
long int atol(const char *nptr);
```

Description

The `atoi` and `atol` functions convert strings to integers. The character string `nptr` is scanned up to the first nondigit character other than an optional leading minus sign (–). Leading white-space characters (spaces and tabs) are ignored.

Return values

The `atoi` function returns as an integer the decimal value represented by `nptr`. The `atol` function returns as a long integer the decimal value represented by `nptr`. On the Macintosh, these functions are equivalent because `int` and `long` are the same size.

Note

Overflow conditions are ignored. A plus sign (+) is considered a nondigit character.

See also

`atof`, `scanf`, `strtol`

String–to–`long`-conversion—`strtol`, `strtoul`

Synopsis

```
#include <StdLib.h>
long int strtol(const char *nptr, char **endptr, int base);
unsigned long int strtoul(const char *nptr, char **endptr, int base);
```

Description

The `strtol` function returns a `long` containing the value represented by the character string `nptr`. The string is scanned up to the first character inconsistent with the base (decimal, hexadecimal, or octal). Leading white-space characters are ignored.

If the value of `endptr` is not `NULL`, a pointer to the character terminating the scan is returned in `*endptr`. If no integer can be formed, `*endptr` is set to `nptr` and 0 is returned.

If `base` is `0`, the base is determined from the string. If the first character after an optional leading sign is not `0`, decimal conversion is done; if the `0` is followed by `x` or `X`, hexadecimal conversion is done; otherwise octal conversion is done.

The function call `atol(nptr)` is equivalent to
`strtol(str, (char **)NULL, 10)`

The function call `atoi(nptr)` is equivalent to
`(int) strtol(str, (char **)NULL, 10)`

The `strtoul` function is similar to the `strtol` function, except that `strtoul` returns an `unsigned long` value.

Note

Overflow conditions are ignored. Apple base conventions (`$` for hexadecimal, `%` for binary) are not supported.

See also

`atof, atoi, scanf`

String–to–`float` conversion—`atof`

Synopsis

```
#include <StdLib.h>
extended atof(const char *nptr);
```

Description

The `atof` function converts a character string pointed to by `nptr` to an extended-precision floating-point number. The first unrecognized character ends the conversion. The function `atof` recognizes an optional string of white-space characters (spaces or tabs), then an optional sign, then a string of digits optionally containing a decimal point, then an optional *e* or *E* followed by an optionally signed integer. If the string begins with an unrecognized character, `atof` returns a NaN.

The `atof` function recognizes `"INF"` as infinity and `"NAN"` (optionally followed by parentheses that may contain a string of digits) as a NaN, with the NaN code given by the string of digits. Case is ignored in the infinity and NaN strings.

Diagnostics

The `atof` function honors the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

See also

`atoi, scanf, strtol`

String to `double` conversion—`strtod`

Synopsis

```
#include <StdLib.h>
double strtod(const char *nptr, char ** endptr);
```

Description

The `strtod` function returns a `double` containing the value represented by the character string `nptr`. The string is scanned up to the first character inconsistent with a floating-point constant. Leading white-space characters are ignored.

If the value of `endptr` is not `NULL`, a pointer to the character terminating the scan is returned in `*endptr`. If no integer can be formed, `*endptr` is set to `nptr` and 0 is returned.

See also

`atoi, scanf, strtol`

Random number functions—`rand, srand`

Synopsis

```
#include <StdLib.h>
int rand(void);
void srand(unsigned int seed);
```

Description

The `rand` function uses a multiplicative congruential random-number generator with period 2^{32} that returns successive pseudorandom numbers in the range from 0 to $2^{15}-1$.

The `srand` function can be called at any time to reset the random-number generator to a specific `seed`. The generator is initially seeded with a value of 1. Identical seeds produce identical sequences of pseudorandom numbers.

See also

Inside Macintosh

Memory-management functions—`malloc`, `free`, `realloc`, `calloc`

Synopsis

```
#include <StdLib.h>
void *malloc (size_t size);
void free (void *ptr);
void *realloc(void *ptr, size_t size);
void *calloc(size_t nmemb, size_t size);
```

Description

The `malloc` and `free` functions provide a simple general-purpose memory allocation package. The storage area expands as necessary when `malloc` is called.

The `malloc` function allocates the first sufficiently large contiguous free space it finds and returns a pointer to a block of at least `size` bytes suitably aligned for any use. It calls `NewPtr` (see *Inside Macintosh*) to get more memory from the system when there is no suitable space already free.

The `free` function takes a parameter that is a pointer to a block previously allocated by `malloc`. If its size is greater than 2K bytes, it is returned to the system using `DisposePtr`. Blocks smaller than that are cached by `malloc` for further allocation by `malloc` only. Undefined results occur if the space assigned by `malloc` is overrun or if a random value is passed to `free`.

The `realloc` function changes the size of the block pointed to by `ptr` to `size` bytes and returns a pointer to the (possibly moved) block. The contents are unchanged up to the lesser of the new and old sizes. If no free block of `size` bytes is available in the storage area, `realloc` asks `malloc` to enlarge the storage area by `size` bytes and then moves the data to the new space. If `ptr` is `NULL`, `realloc` is equivalent to `malloc`.

The `calloc` function allocates space for an array of `nmemb` elements of size `size`. The space is initialized to zeros.

Note

You should not call `DisposePtr` directly to release memory areas allocated with these routines.

Diagnostics

The `malloc`, `realloc`, and `calloc` functions return a null pointer if there is no available memory or if the storage area has been detectably corrupted by storing outside the bounds of a block. When this happens, the block pointed to by `ptr` may have been destroyed.

See also

`setbuf`

Communication with the environment

MPW C provides several functions to facilitate clean termination programs and to access MPW shell variables.

Program termination—`abort`

Synopsis

```
#include <StdLib.h>
void abort (void);
```

Description

The `abort` function requests a program to terminate with error status. It is equivalent to the following program fragment:

```
raise (SIGABRT);
exit (ABORT_STATUS); /* ABORT_STATUS is used for illustration */
                     /* only. Any non-zero value will do */
```

The request will be honored unless

- you install a signal handler for the signal `SIGABRT`, and
- your signal handler doesn't return (that is, it exits by calling `longjmp`).

See also

`signal, raise, longjmp, assert, exit, atexit`

Install a function to be executed at program termination—`atexit`

Synopsis

```
#include <StdLib.h>
int atexit(void (*func)(void));
```

Description

The `atexit` function installs the `exit` function pointed to by `func`, by adding it to a list. The list is initially empty. A list entry is added whenever `atexit` is called. The function `exit` calls the functions in the list in the reverse order to that in which they were added. Programs that use the buffered I/O portion of the Standard I/O Package (including the predefined streams `stdin`, `stdout`, and `stderr`) need to flush all open buffers before the program terminates. To ensure that the buffers are flushed, the Standard I/O Package adds its cleanup function to the list the first time it allocates a buffer.

Each function in the list is called with an `int` parameter either at program termination or when `exit` is called. The argument is the program's status value (zero for normal execution, nonzero for errors). The function can use this value or ignore it.

The number of user-supplied exit functions is limited to 32.

Diagnostics

The function returns a zero value if the installation succeeds.

Note

This function was called `onexit` in previous versions of MPW C.

Warning

If a function is installed more than once, it is executed as many times as it was installed.

See also

`exit, stdio`

Terminate the current application—`exit`

Synopsis

```
#include <StdLib.h>
void exit(int status);
```

Description

The `exit` function closes open file descriptors and terminates the application or tool. Here is the order in which `exit` performs its duties:

1. It executes all exit procedures in reverse order of their installation by `atexit`, including the exit procedures for the Standard I/O Package if Standard I/O routines were used. All buffered files are flushed and closed.
2. It closes all open files that were opened with `open`.
3. If the program is a tool running under the MPW shell, `exit` places the lower three bytes of `status` into the shell's `status` variable and returns control to the MPW Shell.

If the program is an application, `exit` terminates the application.

Return values

The `main` program is a function that returns an integer. The return value of `main` is interpreted by the MPW shell as the program status. When you call `exit`, the `status` parameter is returned to the MPW shell as the return value for the application's main function: 0 for normal execution or a small positive value for errors (typically 1 to 3). A `main` program that returns to the shell without setting `status` to an integer value will appear to be returning a random status.

There is no return from `exit`.

Note

The `exit` function doesn't close files you opened with calls to the I/O routines documented in *Inside Macintosh*.

Don't call `exit` from a desk accessory.

See also

`fclose, atexit, stdio`

Access exported MPW shell variables—`getenv`

Synopsis

```
#include <StdLib.h>
char *getenv(const char *name);
```

Description

The **environment** is the set of exported variables provided by the MPW Shell. The function `getenv` provides access to variables in this set. (See "Variables" in Chapter 5 of the *MPW 3.0 Reference* for the list of standard exported shell variables.)

The `getenv` function searches the environment for a Shell variable with the name specified by `name` and returns a pointer to the character string containing its value. The null pointer is returned if the shell variable is not defined or has not been exported. The shell-variable name search is case insensitive.

Return values

Upon successful completion, a pointer to the value of `name` is returned. If the Shell variable is not defined or not exported, the function returns the null pointer.

For stand-alone applications, which do not run under the MPW Shell, `getenv` always returns the null pointer.

Note

The environment can also be accessed by means of a parameter to the C main-entry-point function `main` if the main procedure is declared as

```
main(argc, argv, envp)
```

The `envp` array represents the set of MPW Shell variables that have been made available to tools by means of the MPW Export command. The `i`th `envp` entry has the form

```
envp[i] = "varname\0varvalue\0";
```

The last `envp` entry is the `null` pointer.

If you use `envp` to search the environment, be sure to use case-insensitive string comparisons.

Warning

The `getenv` function returns a pointer to the place in memory where a copy of the MPW Shell variable resides. Do not modify the value of a Shell variable in such a way as to increase its length.

Communicate with the host system—`system`

Synopsis

```
#include <StdLib.h>
int system (const char *string);
```

Description

The `system` function passes a string to the host environment. This string is then executed by the command processor currently running on the system. To inquire whether a command processor exists, you can pass a null pointer; the `system` function will return a nonzero value only if a command processor is available.

Note

MPW doesn't support commands that are passed from tools. The `system` function always returns 0, signifying that no command processor exists. This function is provided for ANSI compatibility only.

See also

`getenv`

Searching and sorting utilities

The Standard C Library provides utilities for sorting and searching arrays.

Search for an item in a sorted array—`bsearch`

Synopsis

```
#include <StdLib.h>
void *bsearch(const void *key, const void *base, size_t nmemb, size_t
        size,int (*compar)(const void *, const void *));
```

Description

The `bsearch` function searches through an array previously sorted into increasing order. The argument `base` corresponds to the initial array member, the `size` argument corresponds to the size of each array member, and the `nmemb` argument provides the number of members in the array.

The `compar` function is called from `bsearch` with two arguments; the first points to the `key` object, and the second points to an array member. The `compar` function compares array members to `key` until it finds one that is equal to `key`. When a match is found, `compar` returns a zero, and the `bsearch` function returns a pointer to the matching member of the array. If no matching array member is found, `compar` returns a nonzero value, and `bsearch` returns a null pointer.

See also

`qsort`

Quicker sort—`qsort`

Synopsis

```
#include <StdLib.h>
void qsort(void *base, size_t nmemb, size_t size, int (*compar)
        (const void *, const void *));
```

Description

The `qsort` function is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

The `base` parameter points to the element at the base of the table. Parameter `nmemb` is the number of elements in the table. Parameter `size` is the size of an element in the table; it can be specified as `sizeof(*base)`.

The `compar` parameter is a pointer to a comparison function that you supply. The function `qsort` calls your comparison function with pointers to two elements being compared. Here is a sample prototype for your comparison function:

```
int myCompare(char *elem1, char *elem2);
```

Your comparison function supplies the result of the comparison to `qsort` by returning one of the following integer values.

Result	Meaning
<0	The first parameter is less than the second parameter.
0	The first parameter is equal to the second parameter.
>0	The first parameter is greater than the second parameter.

Note

The `base` parameter, the pointer to the base of the table, should be of type pointer-to-element.

See also

`bsearch`

Integer arithmetic functions

The Standard C Library provides functions for integer arithmetic.

Integer absolute value—`abs`, `labs`

Synopsis

```
#include <StdLib.h>
int abs (int i);
long int labs (long int j);
```

The `abs` function returns the absolute value of `i`.

The `labs` function returns the absolute value of long integer `j`.

Note

The absolute value of the negative integer with the largest magnitude is undefined.

See also

`floor`

Integer division—`div`, `ldiv`

Synopsis

```
#include <StdLib.h>
div_t div (int numer, int denom);
ldiv_t ldiv (long int numer, long int denom);
```

Description

The `div` function divides `numer` by `denom`, computing the quotient and the remainder. The results are stored in the structure `div_t`.

The `ldiv` function divides a long integer value `numer` by the long integer `denom`, computing the quotient and the remainder. The results are stored in the structure `ldiv_t`.

See also

`math.h`

Multibyte character functions—`mblen`, `mbtowc`, `wctomb`

Synopsis

```
#include <StdLib.h>
int mblen (const char *s, size_t n);
int mbtowc (wchar_t *pwc, const char *s, size_t n);
int wctomb (char *s, wchar_t wchar);
```

Description

The `mblen` function computes the size in bytes of the multibyte character pointed to by `s`. This function is similar to

```
mbtowc((wchar_t *)0, s, n);
```

except that the `mbtowc` function doesn't affect the shift state.

The `mbtowc` function also computes the size in bytes of the multibyte character pointed to by `s`. It then determines the corresponding code for that multibyte character. This code has a value of type `wchar_t`. (The null character's code has a value of zero.) For valid multibyte characters when `pwc` is not a null pointer, `mbtowc` stores the code in `pwc`. A maximum of `n` bytes of array `s` will be used. The value returned will never be greater than the `MB_CUR_MAX` macro.

The `wctomb` function computes the number of bytes needed to represent the multibyte character corresponding to the code whose value is `wchar` (including any change in shift state). The `wctomb` function stores the multibyte character representation in array object `s` (if `s` is not a null pointer). A maximum of `MB_CUR_MAX` characters are stored. If `wchar` is zero, `wctomb` remains in the initial shift state. The value returned will never be greater than the `MB_CUR_MAX` macro.

See the section on "Locale" for information on how the multibyte character functions are affected by the LC_CTYPE category of the current locale.

Return values

For the multibyte character functions, the following values may be returned.

If `s` is not a null pointer:

0	Returned value is a null character.
–1	Returned value does not correspond to a valid multibyte character.

Otherwise, the function returns the multibyte character's size in bytes.

If `s` is a null pointer:

0	Multibyte character encodings don't have state-dependent encodings.
nonzero	Multibyte character encodings have state-dependent encodings.

See also

locale, multibyte string functions

Inside Macintosh, Volume V

Multibyte string functions—`mbstowcs, wcstombs`

Synopsis

```
#include <StdLib.h>
size_t mbstowcs ( wchar_t *pwcs, const char *s, size_t n);
size_t wcstombs (char *s, const wchar_t *pwcs, size_t n);
```

Description

See the section on "Localization," earlier in this chapter, for information on how the multibyte string functions are affected by the `LC_CTYPE` category of the current locale.

The `mbstowcs` function takes as an argument a sequence of multibyte characters from array `s`. These characters begin in the initial shift state and are converted into a sequence of corresponding codes. Up to `n` codes are stored into the array pointed to by `pwcs`. Multibyte characters that follow a null character aren't examined or converted. The null character itself is converted into a code with value zero.

The character conversion is similar to the `mbtowc` function, except that the `mbtowc` function doesn't affect the shift state.

Copying between overlapping objects produces undefined results. If the `mbstowcs` function encounters an invalid multibyte character, it returns `(size_t)-1`. Otherwise, `mbstowcs` returns the number of array elements that were modified; it doesn't include any terminating zero code.

The `wcstombs` function takes as an argument a sequence of codes from array `pwcs`. The `wcstombs` function converts these codes into characters beginning in the initial shift state, then stores these characters in array `s`. Conversion stops if a multibyte character exceeds the limit of `n` total bytes or if a null character is stored.

The code conversion is similar to the `wctomb` function, except that the `wctomb` function doesn't affect the shift state.

Copying between overlapping objects produces undefined results. If the `wcstombs` function encounters a code corresponding to an invalid multibyte character, it returns `(size_t)-1`. Otherwise, `wcstombs` returns the number of bytes that were modified; it doesn't include any terminating null character.

Note

The Script Manager uses a different scheme for handling multibyte characters. See *Inside Macintosh,* Volume V, for more details on multibyte character handling.

See also

local, multibyte character functions

Inside Macintosh, Volume V

String handling—String.h

The Standard C Library provides functions for string operations: copying, concatenation, comparison, searching.

String function conventions

The string parameters (`srcStr`, `destStr`, and so forth) and `s` point to arrays of characters terminated by a null character. The functions `strcat`, `strncat`, `strcpy`, and `strncpy` all alter `destStr`. These functions do not check for overflow of the array pointed to by `destStr`.

Memory operations include the functions `memcmp`, `memcpy`, `memmove`, `memchr`, and `memset`. These functions operate efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Example

```
#include <String.h>
static char str[] = "#car#//pe,,,?diem";
char *token;

token = strtok(str,  "#");     /* token points to "car" */
token = strtok(NULL, ",");     /* token points to "//pe" */
token = strtok(NULL, "?");     /* token points to "diem" */
token = strtok(NULL, "?");     /* token is a null pointer */
```

Copying functions—`memcpy`, `memccpy`, `memmove`, `strcpy`, `strncpy`

Synopsis

```
#include <String.h>
void *memcpy (void *dest, const void *source, size_t n);
void *memccpy (void *dest, const void *source, int c, size_t n);
void *memmove (void *dest, const void *source, size_t n);
char *strcpy (char *destStr, const char *srcStr);
char *strncpy (char *destStr, const char *srcStr, size_t n);
```

Description

The `memcpy` function copies `n` characters from memory area `source` to `dest`. It returns `dest`. The function `memmove` also copies characters from `source` to `dest`; it is the same as `memcpy` except that it can also work on overlapping objects.

The `memccpy` function copies characters from memory area `source` into `dest`, stopping after the first occurrence of integer `c` has been copied or after `n` characters have been copied, whichever comes first. It returns either a pointer to the character after the copy of `c` in `dest` or a null pointer if `c` was not found in the first `n` characters of `source`.

The `strcpy` function copies string `srcStr` to string `destStr`, stopping after the null character has been copied. The function `strncpy` copies exactly `n` characters, truncating `srcStr` or adding null characters to `destStr` if necessary. The result is not terminated with a null if the length of `srcStr` is `n` or more. Each function returns `destStr`.

Return values

The `memcpy`, `memmove`, and `memccpy` functions return the value of `dest`. The `strcpy` and `strncpy` functions return the value of `destStr`.

Warning

Unless otherwise noted, overlapping moves yield unexpected results.

Concatenation functions—`strcat`, `strncat`

Synopsis

```
#include <String.h>
char *strcat (char *destStr, const char *srcStr);
char *strncat (char *destStr, const char *srcStr, size_t n);
```

Description

The `strcat` function appends a copy of string `srcStr` to the end of string `destStr`. The `strncat` function appends at most `n` characters. Each function returns a pointer to the null-terminated result.

Return values

The `strcat` and `strncat` functions return the value of `destStr`.

Comparison functions—memcmp, strcmp, strncmp, strcoll, strxfrm

Synopsis

```
#include <String.h>
int memcmp (const void *s1, const void *s2, size_t n);
int strcmp (const char *s1, const char *s2);
int strncmp (const char *s1, const char *s2, size_t n);
int strcoll (const char *s1, const char *s2);
size_t strxfrm (char *s1, const char *s2, size_t n);
```

Description

The `memcmp` function compares its parameters, `s1` and `s2`, looking at the first `n` characters only. It returns an integer less than, equal to, or greater than 0, depending on whether `s1` is less than, equal to, or greater than `s2` in the ASCII collating sequence.

The `strcmp` function performs a comparison of its parameters according to the ASCII collating sequence and returns an integer less than, equal to, or greater than 0 when `s1` is less than, equal to, or greater than `s2`, respectively. The function `strncmp` makes the same comparison but looks at a maximum of `n` characters.

The `strcoll` function compares string `s1` to string `s2` in a locale-specific way. It returns an integer less than, equal to, or greater than 0, depending on whether `s1` is less than, equal to, or greater than `s2` in the current locale's collating sequence. Strings `s1` and `s2` are interpreted according to the `LC_COLLATE` category in the current locale. (The Toolbox function `EqualString` gives a similar result.)

The `strxfrm` function transforms a locale-specific string `s2` into a normal string and places the result into `s1`. It returns the length of the transformed string. A maximum of `n` characters can be placed in `s1`. If you transform two strings by using `strxfrm` and then compare them by using `strcmp`, you will get the same result as if you had compared them (untransformed) by using `strcoll`.

Return values

The `memcmp`, `strcmp`, `strncmp`, and `strcoll` functions return an integer value representing whether `s1` is greater than, less than, or equal to `s2`. The `strxfrm` function returns the length of string `s2`.

Warning

The memcmp, strcmp, and strncmp functions use signed arithmetic when comparing their parameters. The sign of the result will be incorrect for characters with values greater than 0x7F in the Macintosh extended character set.

See also

locale.h

Search functions—memchr, strchr, strpbrk, strspn, strcspn, strstr, strtok

Synopsis

```
#include <String.h>
void *memchr (const void *s, int c, size_t n);
char *strchr (const char *s, int c);
char *strrchr (const char *s, int c);
char *strpbrk (const char *s1, const char *s2);
size_t strspn (const char *spanChrs, const char *srcStr);
size_t strcspn (const char *srcStr, const char *s2);
char *strstr (const char *s1, const char *s2);
char *strtok (char *destStr, const char *tokenStr);
```

Description

The memchr function looks for the first occurrence of integer c in the first n characters of memory area s.

The strchr function looks for the first occurrence of integer c in string s, and the strrchr function looks for the last occurrence of integer c in string s. The null character terminating a string is considered to be part of the string. In previous versions of the Standard C Library, strchr was known as index and strrchr was known as rindex.

The strpbrk function looks for the first occurrence in string s1 of any character from string s2.

The strspn function determines the length of the initial segment of string srcStr that consists entirely of characters from string spanChars.

The strcspn function determines the length of the initial segment of string srcStr that consists entirely of characters not from string s2.

The `strstr` function looks for the first occurence of string `s2` within `s1`.

The `strtok` function considers the string `destStr` as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string `tokenStr`. The first call (with pointer `destStr` specified) returns a pointer to the first character of the first token and writes a null character into `destStr` immediately following the returned token. The function keeps track of its position in the string between calls. Subsequent calls for the same string must be made with a null pointer as the first parameter. The separator string `tokenStr` may be different from call to call. When no token remains in `destStr`, a null pointer is returned.

Return values

The `memchr` function returns either a pointer to the first occurrence of integer `c` in the first `n` characters of memory area `s`, or a null pointer if `c` does not occur.

The `strchr` function returns a pointer to the first occurrence of integer `c` in string `s`, and the `strrchr` function returns a pointer to the last occurrence of integer `c` in string `s`. Each of these functions return a null pointer if `c` does not occur in the string.

The `strpbrk` function returns a pointer to the first occurrence in string `s1` of any character from string `s2`, or a null pointer if no character from `s2` exists in `s1`.

The `strspn` function returns the length of the initial segment of string `srcStr` that consists entirely of characters from string `spanChars`.

The `strcspn` function returns the length of the initial segment of string `srcStr` that consists entirely of characters not from string `s2`.

The `strstr` function returns a pointer to the first occurrence of string `s2` within `s1`, or a null pointer if `s2` doesn't occur.

The `strtok` function returns a pointer to the first character of the first.token, or a null pointer if no token remains in `destStr`.

Example

```
#include <String.h>
static char str[] = "#car#//pe,,,?diem";
char *token;

token = strtok(str, "#");  /* token points to "car" */
token = strtok(NULL, ","); /* token points to "//pe" */
token = strtok(NULL, "?"); /* token points to "diem" */
token = strtok(NULL, "?"); /* token is a null pointer */
```

Miscellaneous functions—`memset`, `strerror`, `strlen`

Synopsis

```
#include <String.h>
void *memset (void *dest, int c, size_t n);
char *strerror (int errnum);
size_t strlen (const char *s);
```

Description

The `memset` function sets the first `n` characters in memory area `dest` to the value of integer `c`. It returns `dest`.

The `strerror` function returns an error message corresponding to the error number `n`. These error messages and their numbers are listed in the section "Errors–ErrNo.h," earlier in this chapter.

The `strlen` function determines the number of characters in `s`, not including the terminating null character.

Return values

The `memset` function returns the value of `dest`. The `strerror` function returns a pointer to an error-message string. The `strlen` function returns the value of `s`.

Date and time—Time.h

The Standard C library provides functions for manipulating and converting the date and time: finding the time, calculating the difference between times, converting to and from string format, and converting between calendar time, local time, and Coordinated Universal Time.

Time-manipulation functions—`clock`, `difftime`, `mktime`, `time`

Synopsis

```
#include <Time.h>
#define  NULL 0
#define size_t unsigned int
#define CLK_TCK 60

/* arithmetic types representing times */
typedef long int clock_t;
typedef unsigned long int time_t;

clock_t clock (void);
double difftime (time_t time1, time_t time0);
time_t mktime (struct tm *timeptr);
time_t time (time_t *timer);
```

Description

The `struct tm` holds the components of a calendar time, broken down as follows.

Component	Meaning	Range (min, max)
`int tm_sec;`	seconds after the minute	(0, 59)
`int tm_min;`	minutes after the hour	(0, 59)
`int tm_hour;`	hours after midnight	(0, 23)
`int tm_mday;`	day of the month	(1, 31)
`int tm_mon;`	months since January	(0, 11)
`int tm_year;`	years since 1900	
`int tm_wday;`	days since Sunday	(0, 6)
`int tm_yday;`	days since January 1	(0, 365)
`int tm_isdst;`	Daylight Saving Time flag	

The `tm_isdst` component is positive if Daylight Saving Time is in effect; zero if it is not in effect; and negative if it is unknown whether it is in effect.

The `clock` function returns the amount of time since the Macintosh was last booted. It returns –1 if it can't determine the time.

The `difftime` function returns the difference between two calendar times, expressed in seconds.

The `time` function returns the current calendar time. It returns a –1 if it can't determine the time.

The mktime function calculates times according to the ranges stored within the struct tm. If successful, mktime uses these ranges to convert the submitted local time into the corresponding calendar time. For example, the local time values for tm_year, tm_mon, and tm_mday will return a calendar time as shown below:

Local time	Calendar time	Comment
tm_year = 88	89	add 14 to 01/88
tm_mon = 14	2	February
tm_wday = 13	6	Saturday

The final value of tm_mday isn't calculated until tm_mon and tm_year are determined. If mktime can't determine the time, it returns a –1.

Time conversion functions—asctime, ctime, gmtime, localtime, strftime

Synopsis

```
#include <Time.h>
char *asctime (const struct tm *timeptr);
char *ctime (const time_t *timer);
struct tm *gmtime (const time_t *timer);
struct tm *localtime (const time_t *timer);
size_t strftime (char *s, size_t maxsize, const char *format,
                 const struct tm *timeptr);
```

Description

The asctime function converts the local time contained in the tm structure into a string of the form

```
Mon    Sep    12 08:08:08 1988\n\0
```

The localtime function coverts the calendar time in seconds to local time contained in struct tm.

The ctime function converts the calendar time returned by timer into a string expressing the local time. This is the same as calling

```
asctime (localtime(timeptr))
```

The gmtime function converts the calendar time returned by timer into Coordinated Universal Time. It returns a NULL.

The `strftime` function includes a number of format characters used to place characters into array `s`. The `strftime format` string can consist of zero or more conversion specifiers and ordinary multibyte characters. Each conversion specifier includes a % preceding one of the characters listed below. All ordinary multibyte characters (including the terminating \0) are copied into array `s` unchanged. If overlapping objects are copied, the behavior is undefined. A maximum of `maxsize` characters can be placed into array `s`. Each conversion specifier is replaced by one of the characters in the following list. The program's locale and the values contained in `timeptr` will determine which characters to use.

Specifier	Replaced by
`%a`	the abbreviated weekday name for the locale
`%A`	the full weekday name for the locale
`%b`	the abbreviated month name for the locale
`%B`	the full month name for the locale
`%c`	the appropriate date and time representation for the locale
`%d`	the day of the month as a decimal number (`01-31`)
`%H`	the hour (24-hour clock) as a decimal number (`00-23`)
`%I`	the hour (12-hour clock) as a decimal number (`01-12`)
`%j`	the day of the year as a decimal number (`001-366`)
`%m`	the month as a decimal number (`01-12`)
`%M`	the minute as a decimal number (`00-59`)
`%p`	the equivalent of either AM or PM for the locale
`%S`	the second as a decimal number (`00-60`)
`%U`	the week number of the year (Sunday as the first day of the week) as a decimal number (`00-53`)
`%w`	the weekday as a decimal number (`0-6`, Sunday is `0`)
`%W`	the week number of the year (Monday as the first day of the week) as a decimal number (`00-53`)
`%x`	the appropriate date representation for the locale
`%X`	the appropriate time representation for the locale
`%y`	the year without century as a decimal number (`00-99`)
`%Y`	the year with century as a decimal number
`%Z`	the time zone name, or no characters if time zone is not determinable
`%%`	the percent sign (`%`)

For any conversion specifier not listed above, the behavior is undefined.

Characteristics of floating types—Float.h

Synopsis

```
#define DBL_DIG           15
#define DBL_EPSILON       (scalb(-52,1.0))
#define DBL_MANT_DIG      53
#define DBL_MAX           (nextdouble(inf(),0.0))
#define DBL_MAX_10_EXP    308
#define DBL_MAX_EXP       1024
#define DBL_MIN           (scalb(DBL_MIN_EXP-1,1.0))
#define DBL_MIN_10_EXP    (-307)
#define DBL_MIN_EXP       (-1021)

#define FLT_DIG           7
#define FLT_EPSILON       (scalb(-23,1.0))
#define FLT_MANT_DIG      24
#define FLT_MAX           (nextfloat(inf(),0.0))
#define FLT_MAX_10_EXP    38
#define FLT_MAX_EXP       128
#define FLT_MIN           (scalb(FLT_MIN_EXP-1,1.0))
#define FLT_MIN_10_EXP    (-37)
#define FLT_MIN_EXP       (-125)

#define FLT_RADIX         2
#define FLT_ROUNDS        ((getround()+1) % 4)

#define LDBL_DIG          19
#define LDBL_EPSILON      (scalb(-63,1.0))
#define LDBL_MANT_DIG     64
#define LDBL_MAX          (nextextended(inf(),0.0))
#define LDBL_MAX_10_EXP   4932
#define LDBL_MAX_EXP      16384
#define LDBL_MIN          (scalb(LDBL_MIN_EXP-1,1.0))
#define LDBL_MIN_10_EXP   (-4931)
#define LDBL_MIN_EXP      (-16382)
```

Description

The header <Float.h> specifies the characteristics of floating types `float`, `double`, and `long double`. The functions `scalb`, `nextfloat`, `nextround`, and `nextextended`, used in the macros above, are declared in <SANE.h>.

See also

Apple Numerics Manual, 2nd edition

Sizes of integral types—Limits.h

Synopsis

```
#define CHAR_BIT      8
#define MB_LEN_MAX    2
#define CHAR_MIN      (-128)
#define CHAR_MAX      127
#define SCHAR_MIN     (-128)
#define SCHAR_MAX     127
#define UCHAR_MAX     255U
#define SHRT_MIN      (-32768)
#define SHRT_MAX      32767
#define USHRT_MAX     65535U
#define INT_MIN       (-2147483648)
#define INT_MAX       2147483647
#define UINT_MAX      4294967295U
#define LONG_MIN      (-2147483648)
#define LONG_MAX      2147483647
#define ULONG_MAX     4294967295U
```

Description

The header <Limits.h> specifies the number of bits in a byte, and the minimum and maximum values for the integral types (that is, `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, and `unsigned long`).

- There are 8 bits in a byte.
- Type `char` is signed, and is 8 bits.
- Types `signed char` and `unsigned char` are 8 bits.
- Types `short` and `unsigned short` are 16 bits.
- Types `int`, `unsigned int`, `long` and `unsigned long` are 32 bits.

Common definitions—StdDef.h

Synopsis

```
#include <StdDef.h>

#define  NULL 0
#define offsetof (structure, field)

typedef int ptrdiff_t;
typedef unsigned int size_t;
typedef short wchar_t;
```

Description

The header <StdDef.h> defines macros `NULL` and `offsetof`, and types `ptrdiff_t`, `size_t` and `wchar_t`. These macros and types may also be defined in several other header files that use them.

The `offsetof` macro tells where a field occurs within a structure. When evaluated, `offsetof` expands to an expression of type `size_t`, which contains the offset in bytes from the beginning of the structure to the designated structure member.

Chapter 4 Using the Macintosh interfaces

THIS CHAPTER EXPLAINS HOW TO CALL FUNCTIONS DEFINED in the five volumes of *Inside Macintosh*—both the toolbox and the Operating System—using the Macintosh interfaces. These functions are available to C, Pascal, and assembly language.

The C header files to the Macintosh interfaces contain the C interfaces to the toolbox and OS functions. They are in the folder {CIncludes}, along with the header files for the Standard C Library. The header files define all the constants and data types and have ANSI-style prototypes for all the functions. ■

Contents

About the Macintosh interfaces 169
- Header files 169
- How the interface is implemented 169
- Parameter types 170
- Passing string parameters 170
- Correspondences between Pascal variant records and C structures 171

The `Str255` type 171

Functions using strings, points, and cells 172

About the Macintosh interfaces

This chapter contains the C definitions of the constants, types, and functions defined in *Inside Macintosh.* The information here is the C equivalent of the Pascal definitions in the Summary section at the end of each chapter of *Inside Macintosh.* Complete documentation for each of the constants, types, and functions defined here is found in the corresponding section of *Inside Macintosh.*

- *Note:* If you have a Macintosh II, you have access to all the routines documented in *Inside Macintosh,* Volumes I through V. If you have a Macintosh SE, a Macintosh Plus, or a Macintosh 512K enhanced—with 128K ROM—you have access to all of the routines documented in *Inside Macintosh,* Volumes I through IV, plus some of those in Volume V. If you have a Macintosh with 64K ROM, you have access only to the routines documented in *Inside Macintosh,* Volumes I through III; you don't have access to those documented in Volume IV or Volume V. Parts of the FixMath library are available in a RAM library for 64K ROM users. Graf3D is available in a library for all users.

This chapter provides an introduction to the Macintosh interfaces.

Header files

The C header files for the Macintosh interfaces are in the {CIncludes} directory. They are divided roughly as the chapters in *Inside Macintosh* (File Manager, Memory Manager, and so forth). To use any of the routines in *Inside Macintosh,* include the appropriate header files in your program.

How the interface is implemented

In MPW 3.0 C, the Macintosh interfaces come in two flavors, both of which are declared in the header files.

The preferred interfaces for MPW 3.0 C use Pascal-style functions spelled as in *Inside Macintosh;* strings are Pascal strings, and points and cells are passed by value. These are the interfaces described in this chapter. (In MPW 2.0 C, these functions were spelled in uppercase.) These interfaces are contained both in the header files, as Pascal-style direct functions, which are substituted in-line by the compiler, and as Pascal-style interface routines in the file {Libraries}Interface.o.

An alternative set of MPW 2.0 C interfaces (for functions that use strings, points, or cells) is found in the file CInterface.o. These functions are spelled in lowercase; with them, strings are C strings, and points and cells are passed by address. These interfaces are described in Appendix D. (In MPW 2.0 C, these functions were spelled in mixed-case.) You may wish to link with both Interface.o and CInterface.o: doing so does not cost you anything, because the linker includes code only for those routines that are actually called.

Parameter types

The MPW 3.0 C, Macintosh interfaces expect small structures, like `Point` or `cell` structures, to be passed by value. String parameters are Pascal-style strings unless otherwise indicated. ResTypes and OSTypes can be expressed as character literals; for example, `'MENU'`. All `VAR` parameters in `extern pascal` declarations must be passed explicitly by address.

Structures are passed to the ROM interface procedures by means of a pointer. The ROM actually expects small structures to be passed by value: that is, if the structure is four bytes or less in size, Pascal calling conventions dictate that the structure should itself be pushed on the stack. When this convention disagrees with what the ROM expects, the ROM conventions are used.

Passing string parameters

There are two kinds of string parameters in MPW 3.0 C: C-style and Pascal strings. In general, C programmers use C strings, which are arrays of characters whose last element is the null byte (`\0`). Functions using this type of string are identified by mixed-case characters and are described below.

The Macintosh ROM, however, expects Pascal strings, which have an initial count byte followed by the string characters (and no null byte at the end).

Because both string formats have an extra byte of information (either a count at the beginning or a null byte at the end), it is possible to transform a string in place from Pascal to C and vice versa. The routines `c2pstr` and `p2cstr` in the library CInterface.o perform these conversions:

```
char *p2cstr(StringPtr aStr);
StringPtr c2pstr(char *aStr);
```

These routines are declared in the header file Strings.h.

The interface routines in CInterface.o do these conversions for you automatically. Whenever you call a mixed-case ROM interface routine and one of the parameters is a string (that is, a pointer to `char`, of type `Str255` as described in *Inside Macintosh*), you should pass a Pascal string. Whenever you call a lowercase ROM interface routine and one of the parameters is a string, you should pass a C string.

Correspondences between Pascal variant records and C structures

Some of the types defined in *Inside Macintosh* that are implemented in Pascal as variant records are implemented in the MPW C header files by means of multiple distinct `struct` declarations. Table 4-1 is a list of such types, with references to *Inside Macintosh* and to the corresponding MPW C header file and C `struct` name.

■ **Table 4-1** Correspondences between variant records and structures

Pascal interface Pascal type	C header file Chapter [volume]	C type	MPW C header file
`ABusRecord`	AppleTalk [2]	`ATLAPRec`	AppleTalk.h
`ABusRecord`	AppleTalk [2]	`ATDDPRec`	AppleTalk.h
`ABusRecord`	AppleTalk [2]	`ATNBPRec`	AppleTalk.h
`ABusRecord`	AppleTalk [2]	`ATDDPRec`	AppleTalk.h
`ParamBlockRec`	File Manager [2]	`CntrlParam`	Devices.h
`ParamBlockRec`	File Manager [2]	`ParamBlockRec`	Files.h
`QElem`	OS Utilities [2]	`QElem`	Osutils.h
`DrvSts`	Disk Driver	`DrvSts`	Disks.h
`DrvSts`	Disk Driver	`DrvSts2`	Disks.h
`Point`	QuickDraw [1]	`Point`	Types.h
`Rect`	QuickDraw [1]	`Rect`	Types.h

The `Str255` type

`Str255` is an array of `unsigned char`, defined as follows.

```
typedef unsigned char Str255[256];
```

Its derivatives (such as `Str63` and `Str27`) are defined similarly. (This is a change from previous versions of MPW C: see Appendix D for details.)

Functions using strings, points, and cells

In MPW 3.0 C, functions using strings, points, and cells behave exactly as described in *Inside Macintosh*. Function names are spelled in mixed case; strings are Pascal strings; and points and cells are passed by value. Most of these interfaces are defined as in-line routines; the rest are in the file Interface.o.

An alternative set of lowercase interfaces, which support MPW 2.0-style calling conventions, is in the file CInterface.o. Its use is explained in Appendix D.

Appendix A **C Compiler Options**

THIS APPENDIX DESCRIBES THE OPTIONS you can use with MPW 3.0's C command, which invokes the MPW C Compiler. ■

Syntax

`C` [*option* ...] [*filename*]

Description

Compiles the specified C source file. Compiling file *filename*.`c` creates object file *filename*.`c.o`. (By convention, C source file names end in a `.c` suffix.) If no filenames are specified, standard input is compiled and the object file c.o is created.

See Chapter 2 for details of the C language definition.

Type

MPW Tool.

Input

If no filenames are specified, standard input is compiled. You can terminate input by pressing Command-Enter.

Output

If you specify the `-e` or `-e2` option, preprocessor output is written to standard output, and no object file is produced.

Diagnostics

Errors and warnings are written to diagnostic output. If the `-p` option is specified, progress and summary information is also written to the diagnostic output.

Status

The following status values may be returned:

0 Successful completion.

1 Errors occurred.

Options

`-b` Generate PC-relative references for functions in the same segment and for string constants (which are kept at the end of the function code module). Useful for writing DA's, WDEF's, and so on.

`-b2` This option implies `-b`, but also allows the code generator to reduce code size by overlaying storage for string constants where possible.

`-b3` Keep string constants in the code segment and overlay them when possible (but always generate A5-relative references for function addresses).

`-c` Don't call the code generator.

`-d` *name* Define *name* to the preprocessor with value 1. This is the same as writing

```
#define name 1
```

at the begining of the source file. (The `-d` option does not override `#define` statements in the source file.)

`-d` *name=string* Define *name* to the preprocessor with value *string*. This is the same as writing

`#define` *name string*

at the begining of the source file.

`-e` Do not compile the program. Instead, write the output of the preprocessor to standard output. This option is useful for debugging preprocessor macros.

`-e2` Do not compile the program. Instead, write the output of the preprocessor to standard output, suppressing all comments. This option is useful for debugging preprocessor macros.

`-elems881` Use in-line MC68881 instructions for all transcendental functions available on the MC68881 processor. See the section "Mathematics—Math.h," in Chapter 3, for a complete list of these functions. This option implies the `-mc68881` option.

`-i` *pathname* [`,`*pathname*]...

Search for include files in the specified directories. Multiple `-i` options may be specified. At most 15 directories will be searched. The search order is as follows:

1. The include-file name is used as specified. If a full pathname is given, then no other searching is applied.

If the file wasn't found, and the pathname used to specify the file is a partial pathname (no colons in the name or a leading colon), then the following directories are searched.

2. The directory containing the current input file.
3. The directories specified in the `-i` options, in the order listed.
4. The directories specified in the shell variable {Cincludes}.

`-m` Generate 32-bit data references. This option allows more than 32K of globals, but produces less efficient code.

`-mbg full` Emit full (untruncated) symbols for MacsBug.

`-mbg on` — Emit full (untruncated) symbols for MacsBug.

`-mbg off` — Don't emit MacsBug symbols.

`-mbg ch8` — Emit MPW 2.0-style (8-bit, special format) MacsBug symbols.

`-mbg` *nn* — Emit MacsBug symbols truncated to *nn* characters maximum.

`-mc68020` — Generate MC68020 instructions whenever doing so would provide faster or smaller object code.

`-mc68881` — Generate MC68881 instructions for all basic floating-point operations.

`-n` — Turn errors arising from incompatible pointer assignments into warnings.

`-o` *objname* — Pathname for the generated object file. If *objname* ends with a colon, it indicates a directory for the output file, whose name is then formed by the normal rules (that is, *inputfilename*`.o`). If *objname* does not end with a colon, the object file is written to the file *objname*.

`-p` — Write progress information (include-file names, function names, and sizes) and summary information (number of errors and warnings, code size, global data size, and compilation time) to diagnostic output.

`-r` — Require prototypes for all called functions. Lack of a prototype causes a warning.

`-s` *name* — Name the object code segment *name*. (The default segment name is Main.)

`-sym full` — Emit object file records containing information for the MPW Symbolic Debugger (SADE™). The object file records can be restricted by one or more of the options `notypes`, `nolines`, or `novars`, in a comma-delimited list after the `-sym on` or `-sym full` option.

`notypes` — Emit object file records containing information for the MPW Symbolic Debugger, but not containing type information.

	`nolines`	Emit object file records containing information for the MPW Symbolic Debugger, but not containing information about which source line a statement came from.
	`novars`	Emit object file records containing information for the MPW Symbolic Debugger, but not containing information about variables.
		An example of this command and its options would be `-sym on,notypes,novars`
	`-sym on`	Same as `-sym full`.
	`-sym off`	Don't emit object file records containing information for the MPW Symbolic Debugger (default).
	`-t`	Write compilation time to diagnostic output.
	`-u` *name*	Undefine the predefined preprocessor symbol *name*. Using this option is the same as writing
		`#undef` *name*
		at the begining of the source file.
	`-w`	Suppress compiler warning messages. (By default, warnings are written to diagnostic output.)
	`-w2`	Emit additional warnings about constructs that the compiler may have reason to suspect. (By default, warnings are written to diagnostic output.)
	`-y` *pathname*	Put the compiler's temporary intermediate (`.o.i`) files in the directory specified by *pathname*.
Example	`C -p Sample.c`	Compiles Sample.c, producing the object file Sample.c.o. Writes progress information to diagnostic output. (Sample.c is found in the CExamples folder.)

Appendix B **Files Supplied With MPW 3.0 C**

MPW 3.0 C IS INTENDED FOR USE WITH THE MACINTOSH PROGRAMMER'S WORKSHOP. The files listed below are on the MPW 3.0 C release disk. The first disk contains the MPW 3.0 C Compiler and library object files. The second disk contains sample programs, the Standard C Library header files, and the Macintosh interfaces. These files may be used directly from the release disk or copied onto a hard disk.

The disk contains files in three directories. The directory CExamples: contains instructions, a makefile, and source for the example programs. The directory CIncludes: contains header files for use with the Standard C Library and Macintosh Interface Libraries. The directory CLibraries: contains library object files. ■

Contents

C Compiler (MPW C1:Tools folder) 181
Object files (MPW C1:Libraries:CLibraries folder) 181
Example source files (MPW C2:Examples:CExamples folder) 181
Library header files (MPW C2:Interfaces:CIncludes folder) 182

C Compiler (MPW C1:Tools folder)

Filename	Description
C	MPW C Compiler

Object files (MPW C1:Libraries:CLibraries folder)

Filename	Description
CInterface.o	C Macintosh Interface Libraries
CLib881.o	MC68881 override library
Complex.o	Complex arithmetic routines
Complex881.o	Complex arithmetic routines for use with MC68881
CRuntime.o	C run-time library
CSANELib.o	C SANE Library
CSANELib881.o	C SANE Library for use with MC68881
Math.o	Mathematical functions library
Math881.o	Mathematical functions library for use with MC68881
StdCLib.o	Standard C Library

Example source files (MPW C2:Examples:CExamples folder)

Filename	Description
Count.c	C source file for Count, a sample MPW tool
Count.r	Rez source file for the Count tool
EditCDev.c	C source file for EditCDev, a sample control device (cdev)
EditCDev.make	Makefile for the EditCDev control device
EditCdev.r	Rez source file for the EditCDev control device
FStubs.c	C source file for dummy library routines
Instructions	Instructions for building examples
MakeFile	Makefile for building examples

Filename	Description
Memory.c	C source file for Memory, a sample desk accessory
Memory.r	Rez source file for Memory desk accessory
Sample.c	C source file for Sample, a minimal MultiFinder-aware application
Sample.h	Header file for the Sampleapplication
Sample.make	Makefile for the Sampleapplication
Sample.r	Rez source file for the Sampleapplication
SillyBalls.c	C source file for SillyBalls, a Color QuickDraw application
SillyBalls.make	Makefile for the SillyBalls application
TESample.c	C source file for TESample, a sample MultiFinder-aware application using TextEdit
TESample.h	C header file for the TESample application
TESample.make	Makefile for the TESample application
TESample.r	Rez source file for the TESample application
TESampleGlue.a	Assembly-language glue source file the TESample application
TESampleGlue.a.o	Object file for the TESample application
TestPerf.c	C source file for TestPerf, a sample program that uses C Performance Tools
TubeTest.c	C source file for TubeTest, a sample application using Color Quickdraw and the Palette Manager
TubeTest.make	Makefile for the TubeTest application
TubeTest.r	Rez source file for the TubeTest application

Library header files (MPW C2:Interfaces:CIncludes folder)

This folder includes the header files for the Standard C Library and the *Inside Macintosh* Libraries. Standard C Library header files, and extensions to the Standard C Library, are so labeled.

Filename	Description
AppleTalk.h	AppleTalk®
Assert.h	Diagnostics (Standard C Library)
Complex.h	Complex arithmetic
Controls.h	Control Manager
CType.h	Character types (Standard C Library)
CursorCtl.h	Animated cursor control
Desk.h	Desk Manager
DeskBus.h	Apple DeskTop Bus™ (ADB) Manager
Devices.h	Device Manager
Dialogs.h	Dialog Manager

Filename	Description
DisAsmLookUp.h	Disassembler package in Toollibs.o
DiskInit.h	Disk initialization
Disks.h	Disk Driver
ErrMgr.h	MPW 3.0 Error Manager
ErrNo.h	Error numbers (Standard C Library)
Errors.h	Macintosh interfaces error numbers
Events.h	Toolbox Event Manager
FCntl.h	Low-level file I/O (Macintosh extensions to the Standard C Library)
Files.h	File Manager
FixMath.h	Fixed-point math
Float.h	Floating-point arithmetic parameters
Fonts.h	Font Manager
Graf3D.h	Graf3D
HyperXCmd.h	HyperCard XCMD interface
IOCtl.h	I/O control (Macintosh extensions to the Standard C Library)
Limits.h	Integer arithmetic parameters
Lists.h	List Manager
Locale.h	Localization (Standard C Library)
Math.h	Mathematical functions (Standard C Library)
Memory.h	Memory Manager
Menus.h	Menu Manager
Notification.h	Notification Manager
OSEvents.h	Operating System Event Manager
OSUtils.h	Operating System Utilities
Packages.h	Packages
Palette.h	Includes Palettes.h
Palettes.h	Color Palette Manager
Perf.h	C Performance Tools
Picker.h	Color Picker interface
Printing.h	Printing Manager header file for backward compatibility
PrintTraps.h	Preferred Printing Manager
QuickDraw.h	QuickDraw
Resources.h	Resource Manager
Retrace.h	Vertical Retrace Manager
ROMDefs.h	Configuration ROM
SANE.h	Standard Apple Numerics Environment
Scrap.h	Scrap Manager
Script.h	Script Manager

Filename	Description
SCSI.h	SCSI Manager
SegLoad.h	Segment Loader
Serial.h	Serial Drivers
SetJmp.h	Non-local jumps (Standard C Library)
ShutDown.h	Shutdown Manager
Signal.h	Signal handler (Standard C Library)
Slots.h	Slot Manager
Sound.h	Sound Drivers
Start.h	Start Manager
StdArg.h	Variable arguments list (Standard C Library)
StdDef.h	Common definitions (Standard C Library)
StdIO.h	Standard I/O (Standard C Library)
StdLib.h	General utilities (Standard C Library)
String.h	String routines (Standard C Library)
Strings.h	Macintosh string conversion routines
SysEqu.h	Low-memory globals
TextEdit.h	TextEdit
Time.h	Date and time (Standard C Library)
Timer.h	Time Manager
ToolUtils.h	Toolbox Utilities
Traps.h	Trap codes
Types.h	Common types
Values.h	Arithmetic values
Video.h	Video drivers
Windows.h	Window Manager

Appendix C Calling conventions and type correspondence

MPW C USES TWO DIFFERENT FUNCTION-CALLING CONVENTIONS: C calling conventions and Pascal-style calling conventions. C-style calling conventions are used for calling functions in the Standard C Library, for example; Pascal-style calling conventions are used for calling functions in the Macintosh interfaces. The conventions differ in the order in which parameters are pushed on and pulled off the stack, among other things. The differences are described here. ■

Contents

C-style calling conventions 187
- C parameters 187
- C function results 187
- C register conventions 188

Pascal-style calling conventions 188
- Pascal parameters 188
- Pascal function results 189
- Pascal register conventions 189

Correspondence between C and Pascal data types 189

C-style calling conventions

This section describes the normal C calling conventions. It explains how function parameters are passed, how function results are returned, and how registers are saved across function calls. This information is useful when you are writing calls between C and assembly language.

C parameters

Parameters to C functions are evaluated from right to left and are pushed onto the stack in the order in which they are evaluated. Characters, integers, and enumerated types are passed as sign-extended 32-bit values. Pointers and arrays are passed as 32-bit addresses. Types `float`, `double`, `comp`, and `extended` are passed as extended 80-bit values (or 96-bit for `-mc68881`). Structures are also passed on the stack. Their size is rounded up to a multiple of 16 bits (2 bytes). If rounding occurs, the unused storage has the highest memory address. The caller removes the parameters from the stack.

◆ *Note:* Programs that depend on right-to-left evaluation are nonportable.

C function results

Characters, integers, enumerated types, and pointers are returned in register D0, using as many significant bits as is required by the type of the result. Types `float`, `double`, `comp`, and `extended` are returned as `extended` values in registers D0, D1, and A0 (or FP0 for `-mc68881`). The low-order 16 bits of D0 contain the sign and exponent bits, register D1 contains the high-order 32 bits of the significand, and register A0 contains the low-order 32 bits of the significand. Structure values are returned by moving the value into a location, the address of which is passed to the routine as if it were the first parameter (that is, on top of the stack before the JSR).

This scheme is different from that in MPW 2.0 C, as explained in Appendix D.

C register conventions

Registers D0, D1, D2, A0, and A1 (and FP0, FP1, FP2, and FP3 for `-mc68881`) are scratch registers that are not preserved by C functions. All other registers are preserved. Register A5 is the global frame pointer, register A6 is the local frame pointer, and register A7 is the stack pointer. Local stack frames are not necessarily created for simple functions, except when debugger symbol information is being generated.

This scheme is different from that in MPW 2.0 C, as explained in Appendix D.

Pascal-style calling conventions

This section describes the MPW C conventions for calling Pascal functions and for calling C functions that use Pascal-style calling conventions. These conventions differ from the normal C calling conventions described earlier in this appendix. This section explains how function parameters are passed, how function results are returned, and how registers are saved across function calls.

Pascal parameters

Parameters to Pascal-style functions are evaluated left to right and are pushed onto the stack in the order in which they are evaluated. Characters and enumerated types whose literal values fall within the range of types `char` or `unsigned char` are pushed as bytes. (This requires a 16-bit word on the stack. The value is in the high-order 8 bits; the low-order 8 bits are unused.) Short values and enumerated types whose literal values fall within the range of types `short` or `unsigned short` are passed as 16-bit values. `int` and `long` values and the remaining enumerated types are passed as 32-bit values. Pointers and arrays are passed as 32-bit addresses.

A Pascal-style function passes SANE types `comp`, `double`, `extended`, and `float` as 80-bit `extended` values (or 96-bit `extended` values if the `-mc68881` compiler option is set). The Pascal compiler, however, passes these types as `extended` by address. (See Table C-1 for the recommended way to pass SANE-type values to Pascal.)

If a structure's size is greater than 4 bytes, it is passed to a Pascal-style function by address; otherwise, it is passed by value.

Pascal function results

Function results are returned on the stack. Stack space for the function result is reserved by the caller before pushing any parameters. Characters and enumerated types whose literal values fall in the range of types `char` or `unsigned char` are returned as bytes. (This requires a 16-bit word on the stack. The value is in the high-order 8 bits; the low-order 8 bits are unused.) `short` values and enumerated types whose literal values fall in the range of types `short` or `unsigned short` are returned as 16-bit values. `int` and `long` values and the remaining enumerated types are returned as 32-bit values. Pointers are returned as 32-bit addresses. Arrays may not be returned as function results. Results of type `float` are returned as 32-bit values. For structures and types `double`, `comp`, and `extended`, the caller pushes the address of a structure, `comp`, `double`, or `extended` (respectively) in the function-result location on the stack. The procedure being called stores the result at this address. The caller removes the function results from the stack.

For structure results, if the Pascal function returns a structure of greater than 4 bytes, the caller pushes a pointer to a result space before pushing any results. If the structure is 4 bytes or less, the caller reserves 2 or 4 bytes on the stack for it.

Pascal register conventions

Registers D0, D1, D2, A0, and A1 (and FP0 through FP3 for `-mc68881`) are scratch registers. Scratch registers are not preserved by Pascal-style functions. All other registers are preserved. Register A5 is the global frame pointer, register A6 is the local frame pointer, and register A7 is the stack pointer.

Correspondence between C and Pascal data types

C and Pascal support different data types. Therefore, when writing a Pascal-style function declaration in C, you must provide a translation of the parameter types and function-result type (from Pascal to C). Often this translation is obvious, but some cases are surprising. For example, in Pascal, variables of type `char` are passed as 16-bit values. Similarly, when a C program and a Pascal program use the same global or external variables, they must use the corresponding data types.

Table C-1 summarizes this translation. Find the Pascal type in the first column. Use the equivalent C type found in the second column. Comments in the table point out unusual cases that may require special attention. This table covers four kinds of variables:

- The variable is a parameter passed by value.
- The variable is a parameter passed by address.
- The variable is the return value from a function.
- The variable has global scope, because it is declared outside a function.

Table C-1 Parameter and result data types

Pascal data type	C equivalent	Comment
Boolean types		
`Boolean`	`Boolean`	`Boolean` is defined in the file Types.h as `enum {false, true};` `typedef unsigned char Boolean;`
`VAR Boolean`	`Boolean *`	In C, `false` is zero and `true` is considered nonzero.
`Boolean` result	`Boolean`	In Pascal, `false` is zero and `true` is one.
`Boolean` global	`Boolean`	
Integral types		
enumeration type (<128 or >255 literals)	`enum`	Use identical ordering of the enumeration literals.
enumeration type (128 to 255 literals)	`short`	Pascal passes enumerations with 128 or more literals as words.
`VAR` enumeration type (<128 or >255 literals)	`enum *`	
`VAR` enumeration type (128 to 255 literals)	`short *`	
enumeration-type result (<128 or >255 literals)	`enum`	
enumeration-type result (128 to 255 literals)	`short`	
enumeration-type global (<128 or >255 literals)	`enum`	

(continued)

■ **Table C-1** (continued) Parameter and result data types

Pascal data type	C equivalent	Comment
enumeration-type global (128 to 255 literals)	`enum`	
`char`	`short`	Pascal passes `char` parameters as 16-bit values.
`VAR char`	`short *`	Pascal stores unpacked `char` types as 16-bit values.
`char` result	`short`	
`char` global	`short`	
`-128..127`	`char`	
`0..255`	`short`	
`integer`	`short`	16-bit signed values.
`VAR integer`	`short *`	
`integer` result	`short`	
`integer` global	`short` or `unsigned short`	
`longint`	`long`	32-bit signed values.
`VAR longint`	`long *`	
`longint` result	`long`	
`longint` global	`long` or `unsigned long`	
Floating-int types		
`real`	`extended *`	Pascal passes `real` parameters as `extended` by address.
`VAR real`	`float *`	
`real` result	`float`	Pascal returns `real` results by value.
`real` global	`float`	
`double`	`extended *`	Pascal passes `double` parameters as `extended` by address.
`VAR double`	`double *`	

(continued)

■ **Table C-1** (continued) Parameter and result data types

Pascal data type	C equivalent	Comment
`double` result	`double`	The caller supplies the address of the `double` result.
`double` global	`double`	
`comp`	`extended *` or `long double *`	Pascal passes `comp` parameters as `extended` by address.
`VAR comp`	`comp *`	
`comp` result	`comp`	The caller supplies the address of the `comp` result.
`comp` global	`comp`	
`extended`	`extended *` or `long double *`	Pascal passes `extended` parameters by address.
`VAR extended`	`extended *` or `long double *`	
`extended` result	`extended` or `long double`	The caller supplies the address of the `extended` result.
`extended` global	`extended` or `long double`	
Pointer types		
^*PType*	*CType* `*`	32-bit addresses.
`VAR` ^*PType*	*CType* `**`	
^*PType* result	*CType* `*` result	
^*PType* global	*CType* `*` global	
Arrays of pointer type		
`ARRAY` OF *PType* (1 or 2 bytes)	`short`	Pascal passes small arrays by value. (*PType* is any Pascal scalar type.)
`ARRAY` OF *PType* (3 or 4 bytes)	`int` or `long`	
`ARRAY` OF *PType* (5 or more bytes)	*CType* `[]`	Pascal passes larger arrays by address. (*CType* is any C scalar type.)

(continued)

■ **Table C-1** (continued) Parameter and result data types

Pascal data type	C equivalent	Comment
`VAR ARRAY` OF *PType*	*CType* `[]`	
`ARRAY` OF *PType* result	—	C does not allow arrays as results: declare a structure containing an array instead.
`ARRAY` OF *PType* global	*Ctype*`[]`	
Structures		
`RECORD` (1 to 4 bytes)	`struct`	Pascal passes small records by value. (`RECORD` and `struct` must be equivalent.)
`RECORD` (5 or more bytes)	`struct *` (equivalent)	Pascal passes larger records by address.
`VAR RECORD` (any size)	`struct *` (equivalent)	
`RECORD` result (any size)	`struct` (equivalent)	The caller supplies the address of the record result.
`RECORD` global (any size)	`struct` (equivalent)	The fields must match in size and alignment.
Sets		
`SET` (1 to 7 elements)	`char`	Pascal passes sets with 1 to 7 elements as bytes.
`SET` (8 to 16 elements)	`short`	Pascal passes sets with 8 to 16 elements as words.
`SET` (≥17 elements)	`struct`	Pascal also passes larger sets by value.
`VAR SET` (1 to 7 elements)	`char *`	
`VAR SET` (8 to 16 elements)	`short *`	
`VAR SET` (≥17 elements)	`struct *`	
`SET` result (1 to 7 elements)	`char`	Pascal returns small sets by value.

(continued)

■ **Table C-1** (continued) Parameter and result data types

Pascal data type	C equivalent	Comment
`SET` result (8 to 16 elements)	`short`	
`SET` result (≥17 elements)	`struct`	The caller supplies the address of the set result.
`SET` global (1 to 7 elements)	`char`	Pascal returns small sets by value.
`SET` global (8 to 16 elements)	`short`	
`SET` global (≥17 elements)	`struct`	
String types		
`STRING`	`Str255`	Defined in file Types.h, as an array of `unsigned char`.
`var STRING`	`Str255`	
`STRING` result	—	C does not allow strings as results: declare a structure containing an array instead.
`STRING` global	`Str255`	

◆ *Note:* The C `struct` type and the Pascal `record` type do not exactly correspond, as C lacks an equivalent to the Pascal `variant record` type. You can see the relationship by comparing the data structures in the Files.h header file with those in the File Manager chapter of *Inside Macintosh,* Volume IV.

Appendix D Differences between MPW 2.0 C and MPW 3.0 C

A NUMBER OF DIFFERENCES EXIST BETWEEN MPW 3.0 C AND MPW 2.0 C. Several of these were necessary in order to provide ANSI-C features. Differences are described below in four sections: Language, Compiler Options, New Files, Standard C Library and Macintosh Interfaces. ■

Contents

Language 197
- Preprocessor 197
- Keywords 197
- Character constants: the newline, carriage-return, and vertical-tab characters 197
- Conversions of integer types 197
- Bit-fields 198
- Pascal-style functions 198
- Direct functions 199
- Function prototypes 199
- Segmentation specifications 199
- Signed characters 200
- Structure results 201
- Structure comparison (here be dragons) 201
- Register use 202
- `switch statements` 203
- Language anachronisms 203
 - Assignment operators 203
 - Initialization 203
 - Structures and unions 204

C compiler options 204

New files 204

Standard C Library 205
- Header files 205
- Signal-handling mechanism 205

Variable arguments 205
Function `cfree` 205
Function `scanf` and `printf` 206
Macintosh interfaces 206
Pascal-style strings (`Str255`) 206
Functions using strings, points, and cells 207
Using C strings with the Macintosh interfaces 208
Converting MPW 2.0 C source code 209

Language

The most important changes to MPW C have been the addition of ANSI features.

Preprocessor

Stringization and tokenization have been added. These features are described in Chapter 2.

Keywords

Three keywords (`const`, `signed`, and `volatile`) from the ANSI draft C standard have been added to MPW C. Chapter 2 contains a complete list of keywords used in MPW 3.0 C.

Character constants: the newline, carriage-return, and vertical-tab characters

In MPW windows or TextEdit, the **newline** character advances the cursor to the left margin on the next line. In C notation, newline is represented by `\n`. The character code for `\n` in MPW 3.0 C is the ASCII value (13) for the carriage-return character. In C notation, the **carriage-return** character is represented by `\r`. The character code for `\r` in MPW 3.0 C is the standard ASCII value (10) for the line-feed character. (These assignments are reversed from most other implementations of C.)

In MPW 2.0 C, the escape characters `\n` and `\r` both represented the carriage-return character (ASCII value 13). This identity often caused duplicate `switch` labels, and conflicts with the ANSI draft C standard. In MPW 3.0 C, `\n` and `\r` have unique values.

Conversions of integer types

MPW C supports these integral types (in addition to the enumerated types):

- `char`, (or `signed char`) and `unsigned char`
- `short`, (or `signed short`) and `unsigned short`
- `long`, (or `signed long`) and `unsigned long`

Conversions between these types can occur either implicitly, as by combining different types in an expression, or explicitly, as by a cast. These conversions are described in Chapter 2.

MPW 2.0 C was unsigned preserving; MPW 3.0 C is value preserving.

Bit-fields

MPW 3.0 C provides both signed and unsigned bit-fields: bit-fields are signed by default. In MPW 2.0 C bit fields were unsigned by default.

The following example gives different results under the two schemes: MPW 3.0 C (using signed bit-fields) sets `i` to –1; MPW 2.0 C (using unsigned bit-fields) sets `i` to 3:

```
struct {int field:2;} x;
int i;
x.field = 3;
i = x.field;
```

Pascal-style functions

All calls documented in *Inside Macintosh* are now declared to be of type `pascal`. This convention allows parameters to be passed directly to the ROM routines in the manner they expect.

The syntax for declaring Pascal-style functions has been modified. The prototype syntax

```
pascal void DrawChar (char c);    /* prototype syntax */
```

replaces the old-style syntax

```
pascal void DrawChar (c)    /* old style Pascal function declaration */
char c;
extern;
```

The compiler recognizes both, but the current interfaces use the prototype syntax. Future compilers will probably require the prototype syntax. The change only affects the declaration of functions using Pascal-style calling conventions, primarily found in header files supplied with MPW.

Direct functions

The syntax for declaring direct functions has been modified. The keyword `extern` has been replaced with an equal sign. This example

```
pascal void PenSize (short width, short height) = 0xA89B;
```

defines a Pascal-style function that is a direct trap to the Macintosh ROM.

The unary `&` (address-of) operator may not be applied to direct functions. Thus it is impossible to create pointers to direct functions or to pass direct functions as parameters.

A list of instructions, in braces, is also permitted:

```
pascal long SetCurrentA5(void) = {0x2E8D,0x2A78,0x0904};
```

Function prototypes

All callable functions are now declared within the CIncludes files. These files use function prototypes to describe all parameters. These parameters are now checked for correct type by the C compiler.

Calls introduced with MPW 2.0 C that could only be used on the Macintosh SE or Macintosh II were available only if the constant `__ALLNU__` was defined. This restriction no longer applies. All functions, including those introduced with MPW 3.0 C, are available without setting special flags.

Segmentation specifications

Segmentation specifications of the form

`#define __SEG__` *name*

are obsolescent. MPW 3.0 C supports them for compatibility's sake, but future versions will not. The newer, recommended, form is this:

`#pragma segment` *name*

This change can be made in several source files at once by using a suitable MPW shell script, like the one below:

```
Set SP " "
Set CaseSensitive 1
Set Exit 0
For file in ≈.c
   Target "{file}"
   Find •
   Replace -c ∞ ∂
      /•([{SP}∂t]*)®1#([{SP}∂t]*)®2define([{SP}∂t]*)®3__SEG__{SP}/ ∂
        "®1#®2pragma®3segment{SP}"
   Close "{file}"
   Echo "{file}"
End
```

This script preserves indention.

Signed characters

Prior to MPW 3.0 C, character literals were unsigned. In the MPW 3.0 C compiler, both variables and literals of type `char` are signed. This change may cause subtle bugs in your existing code. Examine this simple program:

```
main()
{
char c = '∞';

printf("c %d '∞' %d\n", c, '∞');
if (c == '∞')
   puts("c == '∞'");
else
   puts("c != '∞'");
}
```

In MPW 2.0 C, the result is

```
c -80 '∞' 176
c != '∞'
```

because the literal `'∞'` is treated as unsigned, whereas the `char c` variable is signed.

In MPW 3.0 C, the result is

```
c -80 '∞' -80
c == '∞'
```

Structure results

The implementation for functions that return `structs` or `unions` is incompatible between MPW 2.0 C and MPW 3.0 C.

In MPW 2.0 C, each function that returned a `struct` (or `union`, from now on—we'll just say *structure*) automatically allocated some global variable space that was the size of the structure. The function returning the structure filled in its global variable structure with the return values, put the address of its global variable in D0, and returned. There were some problems with the 2.0 approach:

- Since functions that returned structures automatically declared global variables, their use was curtailed in writing CDEF's, WDEF's, desk accessories, or in any other stand-alone code resource.
- This method for returning structures was non-reentrant. If an interrupt was serviced while the structure-returning function was filling in the global variable, and if the interrupt routine called the same routine, the global variable would be garbage when the interrupt service routine returned.

MPW 3.0 C corrects these problems by having the caller pass the address (as the first argument on the stack) to which the result is to be returned. In some cases, this address will be the address of a structure that the function is being assigned to (for example, `foo = function();`) or will be the address of a temporary structure allocated in the calling procedure's stack frame. This method doesn't use global variables and it is reentrant. However, pre-C-3.0 object files that contain functions or calls to functions that return structures are incompatible with C 3.0 object files. If you have the source files for these object files, you should recompile them. If you don't have the source, there is no simple workaround (short of writing interface code in assembly language).

Structure comparison (here be dragons)

To preserve compatibility with MPW 2.0 C, MPW 3.0 C allows you to compare structures. Comparing structures is not recommended in MPW 3.0 C, because it can produce surprising results. (Structure comparison is an Apple extension: it is not required by the ANSI draft C standard.)

When testing for equality, you should **beware of holes in structures:** for example, the structure

```
struct{
char c;
int i;
} a, b;
```

occupies six bytes, and has an undefined byte between the byte used for `c` and the two bytes used for `i`. The implementation does not zero out this unused byte. Therefore the test

```
if (a==b)
```

may fail, even in this case—

```
struct s
{
char c;
int i;
} a  = {'a', 3}, b = {'a', 3};
```

—where the respective fields are identical, because the space between the fields is not initialized.

Because of this variability, the use of structure comparison is deprecated.

Register use

MPW 2.0 C considered registers D2 and FP2 to be permanent registers. In other words, C routines (or C routines written in assembly language) had to save D2 and FP2 on entry and restore them on exit if those registers were changed by the routine. In order to be more compatible with *Inside Macintosh* guidelines, however, the MPW 3.0 C compiler considers registers D2 and FP2 to be scratch registers (see the section "Variable Allocation and Register Use" in Chapter 2 for a list of all scratch registers.)

You could be affected by this change in at least two areas:

- If you have written assembly-language routines that call C routines, make sure that they don't rely on D2's being saved across the call.
- If you link with object files compiled with the 2.0 compiler and some routine in that old object file calls a routine compiled with the 3.0 compiler, the value of D2 could be unexpectedly changed. If you have the source code, it is clearly a good idea to recompile everything with the 3.0 compiler. If you are using a commercial set of libraries written for both Pascal and C that uses Pascal calling conventions, there won't be any problem, because the 2.0 compiler brackets calls to Pascal functions with a save and a restore of D2.

`switch` statements

MPW 3.0 C allows a `switch` statement to be nested within a compound statement. (Earlier versions of MPW C consider such nesting an error.) The following `switch` statement compiles under MPW 3.0 C but not MPW 2.0 C.

```
switch (i) {
   case 1:
      if (j) {
         case 2:
            i = 3;
      }
   }
```

Language anachronisms

Several constructs formerly considered part of the C language are now considered anachronisms. The anachronisms involve assignment operators, initialization statements, and references to structures and unions.

Assignment operators

The `=op` form of assignment operators is obsolete. **They will be interpreted in a way very different from that originally intended when the `=op` form was equivalent to `op=`.**

For instance,

`x =- 5;` is interpreted as `x = (-5);`
`x =* 5;` is interpreted as `x = (*5);` (dereferencing 5)
`x =& p;` is interpreted as `x = (&p);` (returning the address of `p`)

Initialization

The equal sign that introduces an initializer must be present. The statement

```
int i 1;
```

is considered a syntax error.

Structures and unions

References to members of structures and unions must be to the appropriate structure or union. For example, the reference `a.b` is illegal if `b` is not a member of `a`. References to components of nested structures and unions must be fully qualified (that is, all intermediate levels of the reference must be specified). Failure to do so is a syntax error.

C compiler options

Several MPW 2.0 C compiler options have been renamed, replaced by other mechanisms that provide more explicit control, or eliminated. The changes are listed below:

- The `-g` and `-ga` options, which control function names in the object code and the generation of stack frames, have been combined and replaced by `-mbg` *level*.
- The `-q`, `-q2`, `-x6`, `-x55`, `-x149`, `-z127`, `-z84` options are no longer available.

New files

To avoid conflict with the ANSI-specified file Time.h, the original Macintosh file by that name has been renamed Timer.h. Palette.h has been renamed Palettes.h for consistency with other header file names, like Fonts.h, Menus.h, Windows.h. (A Palette.h file still remains in CIncludes, but now simply includes Palettes.h.)

A completely new file for the Notification Manager, Notification.h, has been added. To allow system access, low memory globals in SysEqu.h and trap numbers in Traps.h have been added. In addition, the file HyperXCmd.h has been added to add XCMDs and XFCNs for HyperCard.

There are also a lot of new header files for the new ANSI libraries, such as Assert.h, Float.h, Limits.h, Locale.h. VarArgs.h has been renamed StdArg.h.

Standard C Library

Extensive modifications, based on the ANSI draft C standard's library definition, have been made to the Standard C Library since MPW 2.0. Most of these modifications provide additional capabilities or enhance portability without affecting existing programs. The incompatibilities are outlined in the sections that follow.

Header files

A few macros and functions are now in different header files. This MPW Shell command can be used to locate your favorite macro or function:

```
Search /name/ "{CIncludes}"≈.h
```

For instance, `DebugStr()` was declared in Strings.h and is now in Types.h.

Signal-handling mechanism

The MPW 2.0 C signal-handling mechanism, used for handling Command-Period, has been replaced with the signal-handling functions defined in the ANSI draft C standard. This change will require minor changes to your source files. The section "Signal Handling—Signal.h" in Chapter 3 has an example showing how the signal-handling mechanism is used.

Variable arguments

The macros for accessing variable argument lists, `va_start`, `va_arg`, and `va_end`, have been modified to conform to the ANSI draft C standard. Although the macros have the same names, `va_start` now requires a second parameter. In addition, a `va_list` must be declared in order to use these macros.

Function `cfree`

Function `cfree` is no longer provided. The call `free(ptr)` is equivalent to `cfree(ptr, nelem, elsize)`.

Function `scanf` and `printf`

In compliance with the ANSI draft C standard, MPW 3.0 C no longer accepts some of the `scanf` and `printf` conversion specifiers used in MPW 2.0 C.

Specifically, `%ne`, `%nf`, and `%ng` no longer mean `long double (extended)`. The specifier `%n` means to write the number of characters read or written so far. To indicate `long double` in 3.0, use `%Le`, `%Lf`, or `%Lg`.

In MPW 2.0 C, `%p` meant to read or write a Pascal string; in MPW 3.0 C, it means to read or write a pointer. To read or write a Pascal string in MPW 3.0 C, use `%P`.

Macintosh interfaces

The Macintosh interfaces for MPW 3.0 C differ from those of MPW 2.0 C in several ways. The CIncludes folder now contains several new files. In both old and new files, all usable functions are now declared in the function-prototype style. Besides the normal evolution of bug fixes and new extensions, there were several major changes within the files:

- It is no longer necessary to define the constant `__ALLNU__` in order to access calls available only on the Macintosh SE or Macintosh II.
- Pascal-style strings have been redefined to be an array of `unsigned char`, rather than a structure with a length byte as they had been.
- Much glue was eliminated by the inclusion of multiword in-line machine language in the header declarations.
- Functions requiring special glue to handle string parameters, cells, or points were redefined for greater efficiency and ease of use. Pre-MPW 3.0 sources must use the CCvt script utility to update to the new format!

Pascal-style strings (Str255)

The definition of the `Str255` type (and of its relatives like `Str63` and `Str27`) has changed between MPW 2.0 C and MPW 3.0 C. In C, strings are simply arrays of characters; for consistency, the `Str255` type is now an array of characters as well.

The pre-3.0 definition was a structure with two elements: a byte length and an array of text. The new definition of `Str255` is

```
typedef unsigned char Str255[256];
```

The new definition is not backward compatible with the old version. If you use `Str255`'s in your code or use structures that have `Str255`'s (or derivatives) embedded in them (such as `SFReply`'s), you will have to make source code changes.

Because MPW 3.0 C considers `char*` and `unsigned char*` to be type compatible, you can pass a string literal to a function expecting a `Str255` without casting the string literal to an `unsigned char *` or `StringPtr`.

Here are a few examples of source-code changes.

Pre-3.0 source	3.0 source
`Str255 string;`	`Str255 string;`
`string.length`	`*string or string[0]`
`string.text`	`&string[1]`
`&string`	`string`
`&string.length`	`string`
`SETWTITLE(wPtr,"\pHello");`	`SetWTitle(wPtr,"\pHello");`

Functions using strings, points, and cells

The conventions for spelling Macintosh function names in C have evolved over time. In MPW 1.0 C, strings that were used in C were null-terminated and had to be converted to and from a structure with a length byte in order to be used with Macintosh ROM calls. All structures in C were passed by address. Points and cells, which were optimized to be passed by value in Macintosh routines, had to be dereferenced. In MPW 2.0 C, an alternate set of calls that used strings, points, and cells in their optimal manner were introduced and spelled in uppercase. They allowed strings to be passed directly with the `\p` option and allowed points and cells to be passed directly by value.

These preferred functions are now spelled exactly the same way, including case conventions, as those documented in *Inside Macintosh*. This is fitting, since they are the routines that work as described by that book. These interfaces are contained both in the header files, as Pascal-style direct functions, which are substituted in-line by the compiler, and as Pascal-style interface routines in the file {Libraries}Interface.o.

For compatibility with old source code that used C strings and passed addresses of points and cells, the necessary function definitions and interface code still exist. They are now spelled in the familiar lowercase C style. CInterface.o now comprises only the interface code for these lowercase calls to convert their C strings or dereference their point and cell parameters. (For this reason, most C users will not have to link with CInterface.o.)

A script utility, CCvt, has been provided to convert MPW 2.0 sources to the case sense used in MPW 3.0. It leaves all spelling the same except for using proper upper and lower case, with one exception: `Rename`. The routine that passes a C string to the Macintosh file system call would collide with the standard C `rename` function if it were spelled the same with only case change. To avoid this collision, this version of the call to the Macintosh file system has been renamed: `fsrename`. (CCvt is described at the end of this appendix.)

In the event of precompiled object modules that require linking with older, mixed-case interface code to convert strings or points, it is possible to link also with an older version (that is, pre-3.0) of CInterface.o. The mixed-case calls will not conflict with the uppercase calls of Interface.o or the lowercase calls of the new CInterface.o.

Here are some examples of the differences between pre-3.0 and 3.0 source code. Either example on the left side is equivalent to either example on the right side. The in-line trap call is more efficient than the 2.0-style glue.

Pre-3.0 source	3.0 source
`Point src, dest;` `AddPt(&src, &dest);`	`Point src, dest;` `addpt(&src, &dest); /* 2.0-style glue */`
`Point src, dest;` `ADDPT(src, &dest)`	`Point src, dest;` `Addpt(src, &dest); /* in-line trap call */`

Using C strings with the Macintosh interfaces

If you prefer to pass C strings or if you have written MPW 2.0 code that expects them, you can use the all-lowercase versions of the Macintosh interfaces. These routines do the following:

1. convert the strings from C strings into Pascal strings
2. fix the stack so the parameters conform to Pascal calling conventions
3. call the ROM
4. convert the output strings from Pascal strings back into C strings

If using the lowercase interfaces to routines, you always pass C strings to the routines and receive C strings in return.

- Note: The conversions happen only when the string is a parameter to the routine. If the string is a field of a structure passed to the ROM, then no conversion is performed.

Finally, if you have a Pascal string and want to convert it into a C string (for example, before you call a routine), you can do so with the routine `p2cstr()`. The routine `c2pstr()` converts a C string into a Pascal string. Both conversions are done in place.

- *Note:* If you want to create your own Pascal string, simply preface your string with a byte count, in this way:

```
char *pascalstring = "\005Hello";
```

or

```
char *pascalstring = "\pHello";
```

This puts an extra null byte at the end, but has the following properties:

- `&pascalstring[0]` (or just `pascalstring`) is a pointer to a Pascal string.
- `&pascalstring[1]` is a pointer to a C string.

Of course, in an application, most strings come from resources rather than constants in your program. This allows your application to be altered with the resource editor (for internationalization or customization).

Converting MPW 2.0 C source code using CCvt

To convert C source code from MPW 2.0 standards for functions using strings or points to the MPW 3.0 conventions, a script utility, CCvt, is provided. (It is found in {MPW}Scripts.) CCvt has a `-p`, progress, option and accepts full shell expressions for file name input.

CCvt - converts C programs using MPW 2.0 and earlier headers to using MPW 3.0 headers. CCvt changes parameter names in functions that use the `Point`, `Cell`, or `Str255` types to match the case sense introduced in MPW 2.0. Functions that use glue to convert C strings or points passed by reference will be changed from their original mixed-case spellings to lowercase. Equivalent functions introduced in MPW 2.0 that accept Pascal strings or points passed by value (and therefore need no glue), are converted from their uppercase spelling to mixed case to match *Inside Macintosh* conventions. The CCvt command has this syntax:

`CCvt [-p] [ -c | -y ]` *filename...*

`-p` allows you to see the progress of CCvt as it converts programs.

`-c` allows confirmation of each file before it is processed by CCvt. It brings up a dialog box with Yes, No, and Cancel buttons enabled. The `-c` opton takes precedence over `-y`.

`-y` converts without confirmation. (This is the default condition.)

filename... When a filename or list of filenames is specified, CCvt will accept only those filenames that end with `.c` or `.h`. This is to make sure that only C source code is affected.

If no filename is specified, this command will bring up a dialog box to allow you to select one file for conversion. The selected filename is not checked for `.c` or `.h` extension, so this is a mechanism to convert any other files that may need it.

As a backup, CCvt will keep a copy of the original file in a file named *filename*`.orig`.

CCvt uses Canon and the specialized dictionaries, CCvtUMx.dict and CCvtMxL.dict, to convert function names in the new file. CCvt checks that these resources are available before beginning execution.

The dictionaries must exist in {MPW}Tools.

Glossary

***:** C symbol denoting a pointer type.

active window: The frontmost window on the desktop.

address: A number that specifies a location of an **object** in memory.

alert: A warning or report of an error, in the form of an alert box, sound from the Macintosh's speaker, or both.

alert box: A box that appears on the screen to give a warning or report an error during a Macintosh application.

alias: A different name for the same entity.

A-line instructions: Unimplemented 68000-family instructions, used by the Macintosh to implement Toolbox and Operating System calls.

allocate: To reserve an area of memory for use.

anachronism: A language feature, dating back to the early days of C, that is no longer supported by most C compilers. Anachronisms are **obsolete.**

ANSI C: The C language as described in the *American National Standard for Information Systems—Programming Language C,* still in draft stage. Until final acceptance of the standard, we refer to **draft proposed ANSI C,** which is probably a close approximation to the final language.

ANSI draft C standard: The May 11, 1988 draft of the *Draft Proposed American National Standard—Programming Language C,* which may be accepted about the time this MPW C 3.0 reaches the public.

application: A program (such as the MPW Shell) that can be launched from the Macintosh Finder. An application runs stand-alone and has the file type `'APPL'`.

application heap: The portion of the heap available to the running application program and the Toolbox.

application space: Memory that's available for dynamic allocation by applications.

array: A data structure containing an ordered set of elements.

ASCII: Acronym for *American Standard Code for Information Interchange;* a system of assigning code numbers to letters, numerals, punctuation marks, and control codes.

automatic variable: A dynamic local variable that comes into existence when a function is called or a compound statement scope is entered and disappears when it is exited.

big-endian processor: A processor that puts the most significant byte of a word first. So named after a Lilliputian political dispute.

block: A group regarded as a unit; usually refers to data or memory in which data is stored.

block device: A device that reads and writes blocks of bytes at a time. It can read or write any accessible block on demand.

buffer: A holding area in RAM where information can be stored temporarily.

button: A standard Macintosh control that causes some immediate or continuous action when clicked or pressed with the mouse. See also **radio button.**

C: (1) A programming language originally designed by Dennis Ritchie in 1972 and expanded by the ANSI draft C standard. (2) The command that invokes the MPW C Compiler.

C string: A sequence of characters terminated by a null byte.

C SANE Library: provides mathematical functions and supports floating-point arithmetic. Some of these routines are called through the Standard C Library: they are documented in Chapter 3. The other SANE routines are called through the Macintosh interfaces.

calling conventions: The conventions under which a function is called, parameters are passed to it, and a result, if any, is returned. MPW C supports both C-style and Pascal-style calling conventions.

carriage return (`\r`): A control code generated by the Return key. In MPW C 3.0 `\r` is the standard ASCII value (10) for the line-feed character.

CCvt: An MPW Shell script that converts programs using MPW C 2.0 and earlier headers to programs using MPW C 3.0 headers, changing function names as required.

`char`: An 8-bit character data type whose range is –128 to 127.

character device: A device that reads or writes a stream of characters, one at a time. It can neither skip characters nor go back to a previous character.

class specifier: A keyword, like `register`, that specifies a type's storage class.

closed file: A file without an access path. Closed files cannot be read from or written to.

code resource: A resource that contains a program's code—most commonly a resource of type `'CODE'` (for applications and MPW tools), but other resource types such as `'DRVR'` and `'PDEF'` also contain code.

code segment: An individual `'CODE'` resource, comprising part of the code of a Macintosh application. Segments are loaded in and out of memory by the Segment Loader.

command: In the Standard C Library, a parameter that tells a function which of several actions to perform; in the MPW Shell, a command name and parameters.

command file: A file consisting of executable commands that can be run from the shell. Also called a *script*.

`comp`: A 64-bit SANE data type with signed integral values and one NaN.

compaction: The process of moving allocated blocks within a heap zone in order to collect the free space into a single block.

Compiler option: A symbol placed in the MPW C command line to send an instruction to the Compiler.

control: An object in a window on the Macintosh screen with which the user, by using the mouse, can cause instant action with visible results or change settings to modify a future action. The control is internally represented in a control record.

control character: A nonprinting character that controls or modifies the way information is printed or displayed. In the Apple II computer family, control characters have ASCII values between 0 and 31, and are typed from a keyboard by holding down the Control key while pressing some other key. In the Macintosh family, the Extended Keyboard provides a Control key.

cursor: A 16-by-16-bit image that appears on the screen and is controlled by the mouse; called the *pointer* in Macintosh user manuals.

`__DATE__`: A reserved preprocessor symbol that represents the current date at compile time.

data buffer: Heap space containing information to be written to a file or device driver from an application, or read from a file or device driver to an application.

data fork: The part of a file that contains data accessed via the File Manager.

denormalized number: A nonzero number that is too small for normalized representation.

deprecated: Not recommended. A feature is deprecated if it is **obsolescent**—that is, it may work on the current compiler, but will not work on future compilers.

dereference: To refer to a block by its master pointer instead of its handle.

desk accessory: A "mini-application," implemented as a device driver, that can be run at the same time as an application. Desk accessories are files of type `'DFIL'` and creator `'DMOV'`, and are installed by using the Font/DA Mover.

desktop: The screen as a surface for doing work on the Macintosh.

Desktop file: A resource file in which the Finder stores the version data, bundle, icons, and file references for each application on the volume.

device: A part of the Macintosh, or a piece of external equipment, that can transfer information into or out of the Macintosh.

device driver: A program that controls the exchange of information between an application and a device.

diagnostic output: The file to which MPW tools, including the C Compiler, write error messages and progress information. Diagnostic output appears following the commands being executed in the active window by default, and can be redirected to other files, windows, and selections. In C, diagnostic output is referenced using stream `stderr`.

dialog: Same as **dialog box.**

dialog box: A box that a Macintosh application displays to request information it needs to complete a command, or to report that it's waiting for a process to complete.

direct function: A function that makes a direct trap call to the Macintosh ROM.

directory: A subdivision of a volume that can contain files as well as other directories; equivalent to a folder.

disabled: A disabled menu item or menu is one that cannot be chosen; the menu item or menu title appears dimmed. A disabled item in a dialog or alert box has no effect when clicked.

display rectangle: A rectangle that determines where an item is displayed within a dialog or alert box.

document window: The standard Macintosh window for presenting a document.

`double`: A 64-bit floating-point data type—the IEEE `double` type.

draft proposed ANSI C: The C language as described in the ANSI draft C standard. Probably a close approximation to the final ANSI C.

`__DUMP__`: A reserved preprocessor symbol whose value is a character string consisting of the current filename with its suffix replaced with the string `".DUMP"`. It is used in conjunction with the `load` and `dump` pragmas.

ellipsis character: (1) The ANSI C ellipsis character (. . .)is three periods in the parameter list of a function prototype. (2) The MPW shell ellipsis character (...) is produced by typing Option-semicolon. The two are not interchangeable.

empty handle: A handle that points to a NIL master pointer, signifying that the underlying relocatable block has been purged.

end-of-file: See **logical end-of-file** or **physical end-of-file.**

`enum`: An enumerated data type of 8, 16, or 32 bits, depending on the range of the enumerated literals.

environment: In SANE, consists of rounding direction, rounding precision, exception flags, and halt settings; in MPW, consists of exported variables and signal-handling capabilities.

event: A notification to an application of some occurrence that the application may want to respond to.

event mask: A parameter passed to an Event Manager routine to specify which types of events the routine should apply to.

event queue: The Operating System Event Manager's list of pending events.

event record: The internal representation of an event, through which your program learns all pertinent information about that event.

exception: An error or abnormal condition detected by the processor in the course of program execution; includes interrupts, traps, and floating-point exceptions.

exception vector: One of a table of vectors in low memory that point to the routines that are to get control in the event of an exception.

exit function: A function that is registered with `onexit` for execution when the program terminates.

`extended`: An 80-bit floating-point data type—the IEEE `extended` type; used in C for all intermediate results. When the command line includes the option `-mc68881`, the `extended` type becomes a 96-bit data type. In MPW C, but not other C's, **`extended`** is synonymous with **`long double`**.

external reference: A reference to a routine or variable defined in a separate compilation or assembly.

`__FILE__`: A reserved preprocessor symbol that represents the current filename at compile time.

file: A named, ordered sequence of bytes; a principal means by which data is stored and transmitted on the Macintosh.

file buffered: A buffering style in which characters sent to an output I/O function are queued and written as a block.

file control block: A fixed-length data structure, contained in the file-control-block buffer, where information about an access path is stored.

file descriptor: A file reference number returned by a `creat` or `open` call.

file directory: The part of a volume that contains descriptions and locations of all the files and directories on the volume. There are two types of file directories: hierarchical file directories and flat file directories.

File Manager: The part of the Operating System that supports file I/O.

file pointer: A pointer to the next byte to be read or written in a **stream.**

file type: A four-character sequence in single quotes, specified when a file is created, that identifies the type of file. Examples of file types are `'TEXT'`, `'APPL'`, and `'MPST'`.

`FILE` variable: A variable containing information about a stream, including the file descriptor and buffer size, location, and style.

filename: A sequence of up to 31 printing characters (excluding colons) that identifies a file. See also **pathname.**

Finder: The application that maintains the Macintosh desktop and launches other programs. The Finder is also the default startup application.

fixed-point number: A signed 32-bit quantity containing an integer part in the high-order word and a fractional part in the low-order word.

`float`: A 32-bit floating-point data type—the IEEE `single` type.

Floating-Point Arithmetic Package: A Macintosh package that supports extended-precision arithmetic according to IEEE Standard 754.

floating-point coprocessor (MC68881): A coprocessor on the Macintosh II that provides high-speed support for extended-precision arithmetic.

flush: To write out the contents of a buffer.

font: A complete set of characters of one typeface. A font may be restricted to a particular size and style, or may comprise multiple sizes, or multiple sizes and styles.

font script: The writing system used by the font currently designated by `thePort`; hence the system that determines in what form text characters are displayed to the user.

font size: The size of a font in points; equivalent to the distance between the ascent line of one line of text and the ascent line of the next line of single-spaced text.

fork: One of the two parts of a file; see **data fork** and **resource fork.**

free block: A memory block containing space available for allocation.

function: A C routine, which may or may not return a value. Respectively equivalent to a Pascal function or a Pascal procedure.

function prototype: A function declaration containing an argument list as specified by the draft proposed ANSI standard. Function prototypes, one of the major features of ANSI C, allow stronger type checking by the compiler.

full pathname: A pathname beginning from the root directory. A full pathname is a pathname that contains embedded colons but no leading colon. Compare **partial pathname.**

global variable: A variable that is valid for all applications.

`grafPort`: The data structure for the QuickDraw graphics port. This term is sometimes used to mean *graphics port.*

handle: A pointer to a master pointer, which designates a relocatable block in the heap by double indirection.

Hardware SANE: A SANE implementation that uses makes direct MC68881 calls. It requires a numeric coprocessor and uses it to full advantage.

header file: A file whose contents will be included with the source file at compile time; it contains function declarations, macros, types, and `#define` directives used by the Compiler. Also called an *include file.*

heap: The area of memory in which space is dynamically allocated and released on demand, using the Memory Manager.

heap zone: An area of memory initialized by the Memory Manager for heap allocation.

HFS: See **hierarchical file system.**

hierarchical file system (HFS): A system in which directories are used to hold files, applications, and other directories. HFS is used on hard disks and on 800K disks.

highlight: To display an object on the screen in a visually distinctive way, such as inverting it.

honor thy parentheses: A SANE commandment restricting compiler optimization. The MPW C Compiler honors parentheses in the source code and executes floating-point operations in the order they specify. The compiler will not rearrange these operations according to the commutative, associative, or distributive properties, because floating-point numbers do not have these properties. (This restriction is permitted but not required by the ANSI draft C standard.)

icon: A 32-by-32-bit image that graphically represents an object, concept, or message.

identifier: The name of an **object**, limited to 63 characters.

index: A numeric value that indicates the position of an element in a sublist or array, expressed by a subscript.

infinity: A SANE representation of mathematical ∞.

insertion point: An empty selection range; that is, the character position where text will be inserted (marked with a blinking vertical bar).

`int`: A 32-bit integer data type whose range is –2,147,483,648 to 2,147,483,647. Identical to type `long`.

Integrated Environment: A library of functions that provide low-level I/O, access to font and tab settings associated with text files, and signal-handling capabilities. Many of the routines in the Integrated Environment library are intended for use by MPW tools. In C, the routines are part of the Standard C Library.

interface: The compile-time and runtime linkage between your C program and Pascal routines such as those documented in *Inside Macintosh*.

I/O: Abbreviation for *input* and *output* operations, taken collectively.

item: In dialog and alert boxes, a control, icon, picture, or piece of text, each displayed inside its own display rectangle. See also **menu item.**

jump table: A table that contains one entry for every routine called from another segment in an application or MPW tool, and is the means by which the loading and unloading of segments is implemented.

K&R C: The original version of the C language, as described in the first edition of Kernighan and Ritchie's *The C Programming Language*.

keyboard equivalent: The combination of the Command key and another key, used to invoke a menu item from the keyboard.

`__LINE__`: A reserved preprocessor symbol that represents the current line number in the current source file at compile time.

line buffered: A buffering style in which each line of output is saved for writing as soon as a newline character is written.

little-endian processor: A processor that puts the least significant byte of a word first. So named after a Lilliputian political dispute.

lock: To temporarily prevent a relocatable block from being moved during heap compaction.

locked file: A file whose data cannot be changed.

locked volume: A volume whose data cannot be changed. Volumes can be locked by either a software flag or a hardware setting.

logical block: Volume space composed of 512 consecutive bytes of standard information and an additional number of bytes of information specific to the Disk Driver.

logical end-of-file: The position of one byte past the last byte in a file; equal to the actual number of bytes in the file. Compare **physical end-of-file.**

`long`: A 32-bit integer data type whose range is –2,147,483,648 to 2,147,483,647. Identical to type `int`.

`long double`: In ANSI C, a floating-point type that may, but need not, have more precision and range than the **`double`** type. In MPW C (but not all ANSI C's), **`long double`** is synonymous with **`extended`**.

lowercase interfaces: A version of the **Macintosh interfaces** that supports MPW-C-2.0-style calling conventions. The functions have lowercase names.

Macintosh interfaces: A set of **interfaces** that enable you to access *Inside Macintosh* routines (User Interface Toolbox and Macintosh Operating System) from MPW C.

Macintosh Programmer's Workshop (MPW): Apple's software development environment for the Macintosh family.

`main`: The name of the function that is the entry point for every C program.

Main: The default segment name.

main event loop: In a standard Macintosh application program, a loop that repeatedly calls the Toolbox Event Manager to get events and then responds to them as appropriate.

main segment: The segment containing the main program.

manager: A set of data structures and routines that perform related Toolbox or Operating System functions. For instance, the Window Manager handles the display and manipulation of windows on the Macintosh screen.

master pointer: A single pointer to a relocatable block, maintained by the Memory Manager and updated whenever the block is moved, purged, or reallocated. All handles to a relocatable block refer to it by double indirection through the master pointer.

MC68000: The microprocessor in the Macintosh Plus and Macintosh SE computers (as well as in the original Macintosh and Lisa computers.)

MC68020: The microprocessor in the Macintosh II computer.

MC68030: The microprocessor in the Macintosh IIx computer. It supports the MC68020 instruction set, and contains a subset of the Paged Memory Management Unit.

MC68851: The Motorola 68851 Paged Memory Management Unit (PMMU), an optional integrated circuit that provides full memory mapping in the Macintosh II, including 24- to 32-bit address mapping and virtual memory support.

MC688881: The floating-point coprocessor in the Macintosh II computer. It makes hardware SANE possible.

MC688882: The floating-point coprocessor in the Macintosh IIx computer. It supports the MC68881 instruction set.

memory block: An area of contiguous memory within a heap zone.

Memory Manager: The part of the Operating System that dynamically allocates and releases memory space in the heap.

menu: A list of menu items that appears when the user points to a menu title in the menu bar and presses the mouse button. Dragging through the menu and releasing over an enabled menu item chooses that item.

menu bar: The horizontal strip at the top of the Macintosh screen that contains the menu titles of all menus in the menu list.

menu item: A choice in a menu, usually a command to the current application.

Menu Manager: The part of the Toolbox that deals with setting up menus and letting the user choose from them.

mixed-case interfaces: The preferred version of the **Macintosh interfaces**, using MPW-C-3.0-style calling conventions and following *Inside Macintosh* exactly. The functions are spelled as in *Inside Macintosh*.

modifier key: A key (Shift, Caps Lock, Option, or Command) that generates no keyboard events of its own, but changes the meaning of other keys or mouse actions.

mounted volume: A volume that has been inserted into a disk drive and has had descriptive information read from it by the File Manager.

MPW: See **Macintosh Programmer's Workshop.**

MPW Shell: The application that provides the environment within which the other parts of the Macintosh Programmer's Workshop operate. The Shell combines an editor, command interpreter, and built-in commands.

MPW tool: An executable program (type `'MPST'`) that is integrated with the MPW Shell environment (contrasted with an **application,** which runs stand-alone).

NaN: Not a Number; a SANE representation produced when an operation cannot yield a meaningful result.

newline (\n): A control code that advances the print position or cursor to the left margin of the next output line. In MPW C 3.0 `\n` is the ASCII value (13) for the carriage-return character.

newline character: Any character, but usually carriage return (ASCII code 13), that indicates the end of a sequence of bytes.

newline mode: A mode of reading data where the end of the data is indicated by a newline character (and not by a specific byte count).

nonrelocatable block: A block whose location in the heap is fixed and can't be moved during heap compaction.

normalized number: A floating-point number that can be represented with a leading significand bit of 1.

number class: In SANE, a floating-point number category: zero, normalized, denormalized, infinite, or NaN.

object: An area of memory that can be examined and stored into. It has an **identifier** and an **address**.

object file: A file containing specifications for data and code-module contents, references to other code and data modules, and segmentation. Object files are produced by the Assembler and Compiler, and are passed as input to the Linker.

obsolescent: A feature that may now work, but will not work on future compilers. Using an obsolescent feature usually produces a compiler warning.

obsolete: A feature that worked on earlier compilers, but is not supported on the current compiler. Using an obsolete feature usually produces a compiler error, but it may silently produce erroneous code.

open file: A file with an access path. Open files can be read from and written to.

Operating System: The lowest-level software in the Macintosh. It does basic tasks such as I/O, memory management, and interrupt handling.

output driver: A device driver that receives data via a serial port and transfers it to an application.

package: A set of routines and data types that's stored as a resource and brought into memory only when needed.

Package Manager: The part of the Toolbox that lets you access Macintosh RAM-based packages.

Paged Memory Management Unit (PMMU): The Motorola 68851 Paged Memory Management Unit, an optional integrated circuit that provides full memory mapping in the Macintosh II, including 24- to 32-bit address mapping and virtual memory support.

parameter: An input to a routine.

parameter block: A data structure used to transfer information between applications and certain Operating System routines.

partial pathname: A pathname beginning from any directory other than the root directory. A partial pathname either contains no colons or has a leading colon.

Pascal-style function: A function, written in Pascal, C, or assembly language, that is declared in C using the `pascal` specifier.

Pascal string: A sequence of characters that begins with a length byte and has a maximum length of 255 characters. This format is used by the Pascal Compiler for variables of type `STRING`, and is the default form of string created by the Assembler.

pathname: A series of concatenated directory names and filenames that identifies a given file or directory. See also **partial pathname** and **full pathname.**

path reference number: A number that uniquely identifies an individual access path; assigned when the access path is created.

physical end-of-file: The position of one byte past the last allocation block of a file; equal to one more than the maximum number of bytes the file can contain. Compare **logical end-of-file.**

physical size: The actual number of bytes a memory block occupies within its heap zone.

PMMU: The Motorola 68851 Paged Memory Management Unit, an optional integrated circuit that provides full memory mapping in the Macintosh II, including 24- to 32-bit address mapping and virtual memory support.

pointer: The address of an object. A pointer and an `int` are the same size.

pragma: An implementation-dependent compiler directive introduced by the ANSI-C keyword `#pragma`. By definition, pragmas are not portable.

preprocessor: Part of the C Compiler that provides file inclusion and macro substitution.

predefined symbol (also called **preprocessor symbol**): One of a set of constants defined to be 1, equivalent to writing "`#define symbol 1`" at the beginning of the source file.

purge: To remove a relocatable block from the heap, leaving its master pointer allocated but set to `NIL`.

purgeable block: A relocatable block that can be purged from the heap.

QuickDraw: The part of the Toolbox that performs all graphic operations on the Macintosh screen.

read/write permission: Information associated with an access path that indicates whether the file can be read from, written to, both read from and written to, or whatever the file's open permission allows.

reallocate: To allocate new space in the heap for a purged block, updating its master pointer to point to its new location.

register-based routine: A Toolbox or Operating System routine that receives its parameters and returns its results, if any, in registers.

register variable: An **automatic variable** that is allocated to a register. The `register` specifier tells the compiler to allocate a register to a variable if possible.

regular expressions: A language for specifying text patterns, using a special set of metacharacters.

relocatable block: A block that can be moved within the heap during compaction.

resource: Data or code stored in a resource file and managed by the Resource Manager.

resource attribute: One of several characteristics, specified by bits in a resource reference, that determine how the resource should be dealt with.

resource compiler: A program that creates resources from a textual description. The MPW Resource Compiler is named Rez.

resource data: In a resource file, the data that makes up a resource.

resource description file: A text file that can be read by the Resource Compiler and compiled into a resource file. The Resource Decompiler disassembles a resource file, producing a resource description file as output.

resource file: Common usage for the **resource fork** of a Macintosh file.

resource fork: The part of a file that contains data used by an application, such as menus, fonts, and icons. An executable file's code is also stored in the resource fork.

resource header: At the beginning of a resource file, data that gives the offsets to and lengths of the resource data and resource map.

resource ID: A number that, together with the resource type, identifies a resource in a resource file. Every resource has an ID number.

Resource Manager: The part of the Toolbox that reads and writes resources.

resource map: In a resource file, data that is read into memory when the file is opened and that, given a resource specification, leads to the corresponding resource data.

resource name: A string that, together with the resource type, identifies a resource in a resource file. A resource may or may not have a name.

resource reference: In a resource map, an entry that identifies a resource and contains either an offset to its resource data in the resource file or a handle to the data if it's already been read into memory.

resource type: The type of a resource in a resource file, designated by a sequence of four characters (such as `'MENU'` for a menu).

result: An output from a routine.

root directory: The directory at the base of a file catalog.

SANE Library: A set of routines that provide extended-precision mathematical functions.

script: Same as **command file.**

`__SEG__`: A reserved preprocessor symbol that is used to specify the current segment name at compile time.

segment: One of several parts into which the code of an application may be divided. Not all segments need to be in memory at the same time. Segment names are specified at compile time by means of the `-s` compiler option or the `#pragma segment` *segment-name* preprocessor directive.

Segment Loader: The part of the Operating System that loads the code of an application into memory, either as a single unit or divided into dynamically loaded segments.

selection: A series of characters, or a character position, at which the next editing operation will occur. Selected characters in the active window are inversely highlighted. Also called *selection range.*

server: A node that manages access to a peripheral device.

`short`: A 16-bit integer data type whose range is –32,768 to 32,767.

signal: A software interrupt that causes a program to be diverted from its normal execution sequence.

Software SANE: A SANE implementation that uses 'PACK's 4 & 5. It runs on all machines and takes some advantage of numeric coprocessors.

stack-based routine: A Toolbox or Operating System routine that receives its parameters and returns its results, if any, on the stack.

Standard C Library: A library of constants, data types, and functions that support both formatted and low-level I/O, string manipulation, character classification, memory allocation, and mathematical functions. The MPW C 3.0 Standard C Library has all the functions required by the ANSI draft C standard, as well as some Apple extensions.

standard error: Same as **diagnostic output.**

standard input: The file from which MPW tools generally read input if no filename parameters are specified. Standard input is by default the text entered while the tool is executing, and can be redirected to other files, windows, and selections. In C, standard input is referenced using stream `stdin`.

standard output: The file to which many MPW tools write their output. Standard output appears following the commands being executed in the active window by default, and can be redirected to other files, windows, and selections. In C, diagnostic output is referenced using stream `stdout`.

stringization: An ANSI C feature that changes an identifier into a string literal.

`Str255`: In MPW C 3.0, an array of `unsigned char`. (Changed from 2.0.)

stream: A file with associated buffering.

`struct`: A record data type.

subdirectory: Any directory other than the root directory.

system error ID: An ID number that appears in a system error alert to identify the error.

system heap: The portion of the heap reserved for use by the Operating System.

system resource: A resource in the system resource file.

system resource file: A resource file containing standard resources, accessed if a requested resource wasn't found in any of the other resource files that were searched. Also called the *System file*.

thumb: The Control Manager's term for the scroll box (the indicator of a scroll bar).

`__TIME__`: A reserved preprocessor symbol that represents the current time.

tokenization: An ANSI C feature that concatenates tokens.

tool: A program that runs under MPW. Tools are passed parameters by the Shell, and can perform I/O to windows and selections. The C Compiler, Rez, and Link are all tools.

Toolbox: Same as **User Interface Toolbox.**

Transcendental Functions Package: A Macintosh package that contains trigonometric, logarithmic, exponential, and financial functions, as well as a random number generator.

trap: An exception caused by instruction execution. It arises from process recognition of abnormal conditions during instruction execution or from use of the specific instruction whose normal behavior is to cause an exception.

trap dispatcher: The part of the Operating System that examines a trap word to determine what operation it stands for, looks up the address of the corresponding routine in the trap dispatch table, and jumps to the routine.

trap dispatch table: A table in RAM containing the addresses of all Toolbox and Operating System routines in encoded form.

trap macro: A macro that assembles into a trap word, used for calling a Toolbox or Operating System routine from assembly language.

trap number: The identifying number of a Toolbox or Operating System routine; an index into the trap dispatch table.

trap word: An unimplemented instruction representing a call to a Toolbox or Operating System routine.

type: A kind of memory object characterized by certain storage properties.

type qualifier: A keyword that can modify a type and give it certain charistics, such as permanance or impermanence.

unbuffered: A buffer style that does not use a buffer for I/O; reading and writing is done one character at a time.

unlock: To allow a relocatable block to be moved during heap compaction.

unmounted volume: A volume that hasn't been inserted into a disk drive and had descriptive information read from it, or a volume that previously was mounted and has since had the memory used by it released.

unordered: The result of a comparison involving a NaN, even if the comparison is between two identical NaNs. Unordered is neither greater than, less than, nor equal.

unpurgeable block: A relocatable block that can't be purged from the heap.

`unsigned char`: An 8-bit character data type whose range is 0 to 255.

`unsigned int`: A 32-bit integer data type whose range is 0 to 4,294,967,295. Identical to `unsigned long`.

`unsigned long`: A 32-bit integer data type whose range is 0 to 4,294,967,295. Identical to `unsigned int`.

`unsigned short`: A 16-bit integer data type whose range is 0 to 65,535.

User Interface Toolbox: The software in the Macintosh ROM that helps you implement the standard Macintosh user interface in your application.

version data: In an application's resource file, a resource that has the application's signature as its resource type; typically a string that gives the name, version number, and date of the application.

`void`: A data type used to declare functions that take no parameters or return no value. The `void` type may be used to cast expressions where values are not used. Pointers to void are also allowed.

volume: A piece of storage medium formatted to contain files; usually a disk or part of a disk. A 3.5-inch Macintosh disk is one volume.

volume attributes: Information contained on volumes and in memory indicating whether the volume is locked, whether it's busy (in memory only), and whether the volume control block matches the volume information (in memory only).

volume control block: A nonrelocatable block that contains volume-specific information, including the volume information from the master directory block.

volume name: A sequence of up to 27 printing characters that identifies a volume; followed by a colon (`:`) in File Manager routine calls, to distinguish it from a filename.

window: An object on the desktop that presents information, such as a document or a message. Each window is internally represented in a window record.

window class: In a window record, an indication of whether a window is a system window, a dialog or alert window, or a window created directly by the application.

Window Manager: The part of the Toolbox that provides routines for creating and manipulating windows.

working directory: An alternate way of referring to a directory. When opened as a working directory, a directory is given a working directory reference number that's used to refer to it in File Manager calls.

Index

Cast of Characters

110
$ 141
& operator 47, 58, 61, 62
% character 109, 141, 161
%% character 112, 162
+ operator 110
++ operator 52
- operator 110
-- operator 52
// 37
<< 62
>> 62
^ 62
| 62
~ 62
$ suffix 17

A

%A 162
%a 162
abort 144
abs 150
absolute value functions 91
access shell variables 146
acos 27, 88
actual arguments 53, 54
actual function 56
ALLNU 199
alphabetical character 75
American National Standard for Information Systems–Programming Language C 3
anachronisms, changes to 203
AND (&&) 51, 62
angle brackets 10
anonymous variables 64
ANSI draft C standard 3, 33, 42
applec symbol 39
Apple extensions 34
AppleTalk.h 171, 182
application 13
 building an 13
 compiling an 14
 linking an 15
argc 59
arguments
 actual 53, 54
 formal 53
argv 59
arithmetic, floating-point 52
arithmetic function, basic 26
arithmetic operations 50–51
ASCII collating sequence 155
asctime 161
asin 27, 88
ASR 62
assembly language
 compatibility with Pascal 56
 routines that call C routines 202
 source code 20
assert 74
Assert.h 73, 74, 182
assignment operators, changes to 203
assignment suppression 113
associative law 52
asterisk 109
atan 27, 88
atanh 27
atan2 88
atexit 145
atof 141
atoi 140
atol 140

B

%B 162
%b 162
backspace 34
base parameter 149
basic arithmetic function 26
binary-decimal conversion 67
binary read mode 105
binary stream 105
binary write mode 105
bit-fields 47, 61
 changes to 198
bitwise operations 62
Boolean 190
-b option 23, 62
brackets 77
bsearch 148–149
buffered I/O 101
buffering scheme, I/O 98
buffering standard I/O 99–100
buffer initialization 100
BUFSIZ 107
Build menu 13, 18
byte 164
byte ordering 60

C

%c 162
call chain 94
calling conventions 185–194
calloc 143–144
carriage return character 33, 197
case sensitivity 12, 35, 40
 comparisons 147
 header file names 73
 identifiers 69
C command 14, 173

C compiler 1, 3
C Compiler options 173–178
CCvt 209–210
'CDEF' resource 23
'cdev' resource 23
ceil 91
cells, functions using 172
cell structures 170
CExamples folder 7, 13, 181–182
cfree 205
C function results 186
character constants 34–35
 changes to 197
character handling 75
character input/output 117–120
 getting input 117–120
char * 45
character of pushback 120
character output 119
character % 109
character testing 75–77
char *currency_symbol 84
char *decimal_point 84
char *grouping 84
char *int_curr_symbol 84
char int_frac_digits 85
CHAR_MAX 86
char *mon_decimal_point 84
char *mon_grouping 85
char *mon_thousands_sep 84
char n_cs_precedes 85
char *negative_sign 85
char n_sep_by_space 85
char n_sign_posn 85
char p_cs_precedes 85
char *positive_sign 85
char p_sep_by_space 85
char p_sign_posn 85
char *thousands_sep 84
char type 43, 44, 60
C header files
 to the Macintosh interface 167
 to the OS functions 167
 to the Toolbox functions 167
CIncludes file 54
CIncludes folder 9, 182–184
CInterface.o 6, 170, 181
circumflex (^) 115
clearerr 123, 124
CLib881.o 5, 6, 16, 26, 181
CLibraries folder 9, 181
clock 159–160
close function 20, 129
closing a file 129
code generator 175
'CODE' 11, 13
colon 10, 176
Color Manager 4
Command-Q (quit) 14
Commands folder 9
comma operator 51
comments, line end 37
communication with the environment 144–147
commutative properties 52
compar 149
comparing structures 201–202
comparison functions 155–156
comparisons involving a NaN 66
compiler 12
compiler limitations 62
Compiler options 26
compile time warnings 41
compiling a desk accessory 22
compiling an application 14
Complex881.o 6, 16, 26, 181
Complex.h 182
Complex.o 6, 16, 26, 181
comp type 44, 45, 64, 65
concatenated string, maximum length 37
concatenation functions 155
conditional compilation 38, 39
conditional operator (?) 51
const type qualifier 29, 47–48
control character 75
control function 20
Controls.h 182
conversion
 between integral types 50
 binary to decimal 67
 decimal to binary 35
 formatted input 112–117
 string to double 142
 string to float 141
 string to int 140
 string to long 140–141
conversion characters 110
conversion functions, floating-point to string 92–93
converting MPW 2.0 C source code 209–210
Coordinated Universal Time 159
copying functions 154
cos 27, 88
cosh 27, 89
Count.c 7, 181
Count.r 7, 181
Count tool 18
C parameters 187
C Programming Language, The 3
creat 126–127
creating a file 126–127
C register conventions 188
CRuntime.o 6, 16, 26, 181
CSANELib881.o 5, 6, 16, 26, 181
CSANELib.o 6, 16, 64, 181
C SANE Library 1, 5, 64, 67
C string 36, 170
 using with the Inside Macintosh libraries 208–209
C-style calling conventions 187
ctime 161
C2pstr 170
Ctype 43
CType.h 73, 75–78, 182
CursorCtl.h 182

D

%d 162
data declarations 63
data definitions 63
data types 42–43
 correspondence between C and Pascal 189–194
__DATE__ 38
date 159–162
DEC 60
decimal notation 110
decimal point 35
decimal-to-binary conversion 35
#define 37
defined 39

definitions 165
desk accessory 19
building a 21–22
compiling a 22
creating a 19–22
linking 22
routines 21
DeskBus.h 182
Desk.h 182
Desk Manager 19
destStr 154
device drivers 124
Device Manager 19
Devices.h 171, 182
diagnostic
output 14, 17, 175
specifications 15
Diagnostics 74
dialogs 13
Dialogs.h 182
difftime 159–160
digit character 75
direct function 57–58
changes to 199
definition 58
Directory menu 13
DisAsmLookUp.h 183
DiskInit.h 183
disksname 13
Disks.h 171, 183
distance function 92
distributive properties 52
div 150
dividend 51
division of int 4
do statement 51
double 43
double quotation marks 10
double type 64
driver glue 21
'DRVR' resource 20
DRVRClose 20, 21
DRVRControl 20, 21
DRVROpen 20, 21
DRVRPrime 20, 21
DRVRRuntime library 20–21
DRVRRuntime.o 6, 20, 22
DRVRStatus 20, 21
__DUMP__ 38
dump 41
dumping and loading 41
dup 136
characteristics of 163

E

EACCES 80
EBADF 80
EBUSH 80
ecvt 92–93
EditCDev.c 7
EditCDev.make 7, 181
EditCDev.r 7, 181
EDOM 81
EEXIST 80
EFAULT 80
EFBIG 81
8086 60
8087 64
EINTR 79
EINVAL 80
EIO 79
EISDIR 80
-elems881 option 4, 25, 27, 63, 65
#elif 39
ellipsis character 34–35
EMFILE 81
EMLINK 81
empty string constant ("") 83
ENFILE 81
ENODEV 80
ENOENT 79
ENOMEM 80
ENORSRC 79
ENOSPC 81
ENOTBLK 80
ENOTDIR 80
ENOTTY 81
enumerated types 46
enum type 3, 45, 60
environment 147
envp 59, 147
ENXIO 79
EOR 62
EPERM 79
equal to (==) 66
ERANGE 81
EROFS 81
ErrMgr.h 183
ErrNo.h 73, 78–81, 183
error-handling 123–124
errors 78–81
ESPIPE 81
E2BIG 80
ETXTBSY 81
evaluation order 52
Events.h 183
Examples folder 9
Example source files 181–182
exception flags 67, 68
floating-point 88
EXDEV 80
exit 101, 145–146
calling from desk accessory 146
exp 27, 89–90
exp1 27
exp2 27
exponent field 35
exponential functions 89–90
exponential notation 35
extended and long double
types, relationship between 65
extended data format 27
extended type 43, 64
extensions, Apple 34
extern 20, 63
external definitions, multiple 63

F

fabs 27, 91–92
faccess 133–135
fclose 101, 104–105
fcntl 136–137
FCntl.h 73, 126, 183
fcvt 92–93
F_DELETE 134
fdopen 105–106
feof 123–124
ferror 123–124
fflush 104–105
fgetc 117–118
fgetpos 122–123
fgets 118
F_GFONTINFO 134
F_GPRINTREC 134
F_GSELINFO 135
F_GTABINFO 134
F_GWININFO 135
__FILE__ 38

file
access functions 104–108
buffering 99, 106–108
closing a 104–105
control and status information 133–135
creating a 126–127
descriptor 127, 128
duplicate a 136–137
duplicate an open descriptor 136
new and renamed 204
opening 105–106, 127–129
operations on 102–104
positioning 122–123, 132–133
reading from a 130–131
removing a 129–130
writing to 131–132
File Manager 169, 194
fileno 123–124
Files.h 171, 183
File variables 17
Finder™ 13, 95
FIOBUFSIZE 125
FIOFNAME 125
FIOINTERACTIVE 125
FIOREFNUM 125
FIOSETEOF 125
FixMath library 169
FixMath.h 183
flag characters 110
float data type 3, 43, 64
Float.h 73, 163, 183
floating-point
chips 64
enhancements 4
exception flags 88
number 114
parameters 64
registers 27
to-string conversion functions 93
types 45
variable 61
floating-point arithmetic 52
speeding up 25
Floating-Point Arithmetic Package 65
floor 91
fmod 91
following F 35
Font/DA Mover 22
Fonts.h 183
font size 135
F_OPEN 134
fopen 105
-f option 63
formal arguments 53
format characters 113
formatted input, converting to 112–117
formatted input/output 108–117
formatted output, printing 108–112
for statement 51
fprintf 108–112
fputc 119
fputs 120
fread 121–122
free 143–144, 205
F_RENAME 134
freopen 105–106
frexp 89–90
fscanf 112–117
fseek 122–123
fsetpos 122–123
F_SFONTINFO 134
F_SPRINTREC 134
F_SSELINFO 135
F_STABINFO 134
FStubs.c 7, 181
F_SWININFO 135
ftell 122–123
full expression 51
full pathname 10
function declarations 53–54
Pascal style 55
function prototype 53
changes resulting from 199
functions
definition 53–54
implemented as macros 99
Pascal style 55
SANE types in 66
functions using strings, points, and cells 170
changes to 207
fwrite 121

G

general utilities 139–159
generic pointers 45
getc 117
with stream parameter 118
getchar 117
getenv 147
gets 118
getting character input 117–118
getw 117
global data area desk accessory 20
global scope 190
global variables 23, 61
gmtime 161
Graf3D.h 183
graphic character 75
greater than (>) 66

H

%H 162
halt
disabled 68
enables 67
hardware-dependent memory locations 48
hardware interrupt 95
difference between signal and 95
hardware SANE routines 64
header files 9, 24, 169
changes to 205
names 73
here be dragons 201
hexadecimal integer 113
hex digit character 75
holes in structures 201
honor thy parentheses 52
host-system communication 147–148
hyperbolic functions 89
HyperXCmd.h 183
HyperXLib.o 6
hypot 92

I

%I 162
IBM 60
identifiers 35
case sensitive 69

IEEE arithmetic 68
IEEE Standard for Binary Floating-Point Arithmetic 64
if statement 51
impermanence 47
implementation
 defined 33
 details 59–63
important variables 95
#include 37
include files 9
INF 67
infinities 64, 65, 68
infinity, signed 90
initialization, changes to 203
'INIT' resource 23
input
 specifications 15
 stream 115
input/output
 buffering MPW C stdio 99–100
 control 124–126
 direct 121
 formatted 108–112
 numeric 66
Inside Macintosh, Volume V 153
installing MPW C 9
Instructions file 7, 181
integer
 absolute value 150
 arithmetic 151
 constant, overflow 36
 conversions 49–50
 division 151
 division, remainder 51
integral constant expression 46
integral types 44
 sizes of 164
Intel 60, 64
interactive interrupt monitoring 95
Interface.o 6, 16
Interfaces folder 9
int type 42, 44
 divisions of 4
 multiplications of 4
I/O buffering scheme 98
ioctl 124–125
IOCtl.h 73, 124–125, 183
_IOLBF 101
-i option 10
_IOSYNC 101
isalnum 75, 76
isalpha 75, 76
isascii 76
iscntrl 75, 76
isdigit 75, 76
isgraph 75, 76
islower 75, 76
isprint 75, 76
ispunct 75, 76
isspace 75, 76
isupper 75, 76
isxdigit 75, 76

J

%j 162
jumps, non-local 93–95
jump table desk accessory 20

K

K&R C 1, 32, 44, 64
K&R 1 29
K&R 2 29
keywords 34
 changes to 197

L

labs 150
language
 changes to 197–198
 definition 33
language anachronisms 203–204
LC_ALL 82
LC_COLLATE 82
LC_CTYPE 82, 151
LC_MONETARY 82
LC_NUMERIC 82
LC_TIME 82
ldexp 89
ldiv 150
leading spaces 12
least-significant bits 51
length byte 36
less than (<) 66
lexical conventions 34–37
Libraries folder 9
Library header files 182–184
Limits.h 73, 164, 183
__LINE__ 38
line buffered 99
line-end comments 37
line-feed character 34
Link
 linker 12, 15
 -m option 23
 -rt option 23
 -sg option 23
linking between 2.0 and 3.0 object files 202
Lists.h 183
literal matches 116
ln1 27
load 40
localeconv 83–87
Locale.h 73, 82–87, 183
localization 82
localtime 161–162
log 27, 89
logarithmic functions 89
logical AND (&&) 51
logical OR (||) 51
log2 27
log10 27, 89
long data type 43, 44
 and extended types, relationship between 65
long double data type 3, 44, 64
longjmp function 48, 94
lowercase character 75
lowercase interfaces 170, 172
low-level file I/O 126–137
lseek 132
LSL 62

M

%M 162
%m 162
Macintosh extended character set 34, 77, 156
Macintosh 512K 24, 169
Macintosh interfaces 1, 4, 5, 167–172
 C and Pascal flavors 169
 changes to 206
Macintosh Plus 24, 169

Macintosh SE 24, 169
macintosh symbol 39
Macintosh II 24, 65, 169
Macintosh XL 24
macros 37
MacsBug 62, 176
 symbol 61
Main default segment 11, 40
main entry point 21
main program function 16, 17, 146
 parameters to 58
MakeFile 7, 13, 21, 181
malloc 44, 143
Math881.o 5, 6, 16, 26, 181
mathematics 82–93
Math.h 73, 87–93, 176, 183
Math.o 6, 16, 26, 181
mblen 151
mbstowcs 152
mbtowc 151
MC68881
 enhancements 25–26
 floating-point coprocessor 26
 instructions 27
 processor 4, 25, 64
-mc68881 option 4, 25, 16, 41, 64
 linking rules 26
mc68881 symbol 39
MC68000 60
MC68000 symbol 39
MC68030 60
MC68020
 enhancements 4, 27
 processor 25, 60
-mc68020 option 4, 25, 27
memccpy 154
memchr 153, 157–158
memcmp 153, 155–156
memcpy 153, 154–155
memmove 153, 154–155
Memory.c 8, 21, 182
Memory desk accessory 21
Memory.h 183
memory management 143–144
Memory Manager 169
Memory.r 8, 21, 182
memset 153, 158–159
menus 13
Menus.h 183
mixed-case
 characters 170
 interfaces 170
mktime 159–160
mode, binary read 105
mode, binary write 105
mode parameter 105
modf 89
monetary quantities, formatting for C 87
-m option 62
Motorola 60, 64
MPW C Compiler 3
MPW C version 3.0 3
MPW Shell 13
MPW 3.0 C 195
MPW tool 17–18
MPW 2.0 C 195
MPW Types.h 21
m68k symbol 39
multibyte
 character functions 151–152
 string functions 152–153
MultiFinder 13, 22
multiple prototypes 53
multiple 680x0 instructions 57
multiplications of int 4

N

\n 34
NAN 67
NaN
 code 65, 90, 114
 comparisons involving a 66
National Semiconductor 60
%ne 206
nearest integer functions 91
newline character 34, 118, 197
%nf 206
%ng 206
non-ANSI extensions
 input/output control 124
 low-level file I/O 126
non-local jumps 93
nonprototype
 declaration 54
 form 53
-n option 41
NOT 62
Notification.h 183
NS16000 60
NULL 59, 81, 165
null
 byte 36
 character (\0) 108
 string constant 83
 terminator 36
numeric
 constants 35–36
 coprocessor 64
numerics environment 66

O

O_APPEND 128
O_BINARY 128
object files 181
O_CREAT 128
octal integer 113
O_EXCL 128
offsetof 165
open function 20, 127–128
opening a file 127–129
open stream 108
Operating System 167
operations on files 102–104
OR (||) 51, 62
O_RDONLY 128
O_RDWR 128
O_RSRC 128
OSEvents.h 183
OSTypes 170
OSUtils.h 171, 183
O_TRUNC 128
output specifications 15
overflow
 integer constant 36
 variables 36

P

\p 36
%p 162, 206
Packages.h 183
Palette.h 183
Palettes.h 183
parameter types 170
parentheses, honoring 52
partial pathname 10
Pascal

compatibility with assembly language 56
function results 189
parameters 188
register conventions 189
strings 170
`pascal` 29
declaration 17
keyword 23
type qualifier 3, 50
type-specifier 55
Pascal-style
calling conventions 188–189
changes to functions 198
function declarations 55
function definitions 55–56, 57
function pointers 57
strings 36
PDP-11 60, 64
Perf.h 183
PerformLib.o 6
period 109
permanence 47
`perror` 123–124
Picker.h 183
plus sign 140
pointer to `char` 170
pointer to `void` 45
pointer types 45
points, functions using 172
Point structures 170
`-p` option 14
`pow` 90
`power` 90
power functions 90–91
pragmas 40–42
`#pragma` segment directive 11, 37
precompiled information 41
predefined symbols 39
preprocessor 37–41
changes to 197
symbols 38
`prime` function 20
`printf` 108–112
changes to 206
printing character 75
printing formatted output 108–112
Printing.h 183
PrintTraps.h 183
prototype declaration 54
prototypes, multiple 53
pseudorandom numbers 142
`ptrdiff_t` type 45, 165
`p2cstr` 170
punctuation mark 75
`putc` 119
`putchar` 119
`puts` 120
`putw` 119
`pwcs` 152

Q

`qsort` 149
QuickDraw.h 183
quicker-sort 149
Quit (Command-Q) 14

R

`\r` 34
`raise` 97–98
RAM, recommended 24
`rand` 142–143
random number generation 142–143
`read` 130
reading from a file 130
`realloc` 143
`register` 29
`register` class specifier 50
register use 60
changes to 202
remainder functions 91
remainder on integer division 51
`remove` 102
removing a file 102, 129
`rename` 102
ResEdit 13
reserved symbols 38
resource compiler 13
resource editor 13
resources
`'CDEF'` 23
creating 13
`'LDEF'` 23
`'MDEF'` 23
noncode 13
`'WDEF'` 23
`'XCMD'` 23
Resources.h 183
ResTypes 170
Retrace.h 183
`return` statement 51
`rewind` 122–123
Rez 13
right shift of negative signed integer 51
ROM code 24
ROMDefs.h 183
`-r` option 53
rounding direction 67
rounding precision 67

S

`%S` 162
Sample application 13
Sample.c 8, 14, 182
Sample.c.o 14, 178
Sample.h 8, 13, 182
Sample.make 8, 13, 182
Sample.r 8, 13, 182
SANE 63
arithmetic 66
expressions 66
library 67
types 45
SANE.h 64, 183
scalar data types 42–45
`scanf` 112–117
changes to 206
scanset format 114
Scrap.h 183
scratch registers 58, 61
Script.h 183, 184
Script Manager 44, 152
search for an item in a sorted array 148
search functions 157–158
searching 145–149
`__SEG__` 12, 38
SegLoad.h 184
segment 11
segmentation 40
segmentation control 11–12
segmentation specifications, changes to 205
Segment Loader 11
`segment` pragma 40
sending a signal 97–98
sequence points 51

Serial.h 184
setbuf 106–108
setjmp function 48, 94
SetJmp.h 73, 93–95, 184
setlocale 82–83
setvbuf 106–108
Shell variable 147
shell variables 146–147
short type 42, 44, 58
ShutDown.h 184
side effects 52
SIGABRT 144
SIGINT 95
signal 96–97
signal, difference between hardware interrupt 95
Signal.h 73, 95–98, 184
signal handling 95–98
signal-handling mechanism, changes to 200
signals 95
signed characters, changes to 200
signed decimal notation 110
signed infinity 90
SillyBalls.c 8, 182
SillyBalls.make 8
sin 27, 88
single data type 3, 43
sinh 27, 89
6502 60
size 107
size_t type 45, 165
Slot Manager 4
Slots.h 184
Software SANE 65
routines 64
-s option 11
sorting 148–149
Sound.h 184
source code compatibility 24
source files 37
space 110
space characters 33
sprintf 108–112
sqrt 27, 90–91
srand 142–143
srcStr 154
sscanf 112–117
stand-alone code resources 23
Standard Apple Numerics Environment (SANE) 3, 63
Standard C Library 1, 5
changes to 205–206
standard input 15, 130
Standard Input/Output package 98–124
standard I/O buffering 99–100
standard open files 100
standard output 17, 132
Start.h 184
static data 93
static variable 20, 61
status function 20
status values 15
StdArg.h 138–139, 184
StdArgs.h 73
__STDC__ 38
StdCLib.o 6, 16, 26, 181
StdDef.h 73, 165, 184
stderr 17, 99, 100
stdin 17, 100, 118
StdIO.h 73, 98–124, 184
StdLib.h 139–153, 184
stdout 18, 100
strcat 153, 155
strchr 157–158
strcmp 155–156
strcoll 155–156
strcpy 153, 154
strcspn 157–158
strerror 158–159
strftime 161–162
string comparisons, case sensitive 147
string concatenation 37
string constant 36–37
empty 83
null 83
string function conventions 153–154
String.h 73, 153–159, 184
string handling 153–159
string input, getting 118
stringization 41
string literals 42
string output 120–121
string parameters, passing 170–171
strings 13
functions using 172
string to double conversion 142
string to float conversion 141
string to int conversion 140
string to long conversion 140
strlen 158–159
strncat 153, 155
strncmp 155–156
strncpy 153, 154–155
strpbrk 157–158
strspn 157–158
strstr 157–158
strtod 142
strtok 157–158
strtol 140–141
strtoul 140–141
Str255 type 171
struct 171
struct declarations 171
structure
assignment 46
changes to 201
comparison 201–202
results 63
structures 46
structures and unions, changes to 204
strxfrm 155–156
Stubs.o 6, 19
suppressed assignments 116
switch statement 51, 62, 203
changes to 203
-sym 41
Symbolic Debugger (SADE) 177
syntax error 53
SysEqu.h 184
system 147–148
System/370 60

T

tan 27, 88
tanh 27, 89
tell 133
temporary file
creating a 103
naming a 103–104
termination
of current application 145–146
with error status 144
installing function for 145
TESample.c 8, 182
TESampleGlue.a 8, 182
TESampleGlue.a.o 8, 182

TESample.h 8, 182
TESample.make 8, 182
TESample.r 8, 182
TestPerf.c 8, 182
TextEdit.h 184
TextEdit interface 4
text stream 105
__TIME__ 38
time 159–162
time
 conversion 161
 manipulation 159–160
Time.h 73, 159–162, 184
TIOFLUSH 125
tm_hour 160
tm_isdst 160
tm_mday 160
tm_min 160
tm_mon 160
tmpfile 103
TMP_MAX 103
tmpnam 103–104
tm_sec 160
tm_wday 160
tm_yday 160
tm_year 160
toascii 77–78
tokenization 42
tokenStr 157
_tolower 77
tolower 77–78
tool 17
 building a 18
 compiling a 18
 creating a 17–18
 linking a 19
toolbox 167
ToolLibs.o 6
Tools folder 9, 181
ToolUtils.h 184
_toupper 77
toupper 77–78
trailing spaces 12
transcendental function 26
Transcendental Functions Package 65
Traps.h 184
trigonometric functions 88
truncation of a field 111
TubeTest.c 8, 182
TubeTest.make 8, 182
TubeTest.r 8, 182
two's complement bit patterns 62
type conversion
 changes to 197
type correspondence 185–194
type qualifiers 47–50
types 43–47
Types.h 171, 184

U

%U 162
unbuffered 99
underscore characters 38
undoing an input 120–121
ungetc 120–121
union, accessing member of different type 46
unlink 129–130
UnloadSeg 11
unsigned
 char* 45
 char type 43, 44
 int type 43, 44
 long type 43, 44
 short type 43, 44, 58
unsigned
 decimal integer 113
 decimal notation 110
 hexadecimal notation 110
 octal notation 110
 -preserving 50
unused pragma 41
unused variables and parameters 41–42
uppercase character 75

V

va_arg macro 138, 205
va_end 138, 205
va_list 138, 205
value-preserving 50
Values.h 184
VARBoolean 190
variable
 allocation 61
 arguments 138–139
 byte alignment 60
 changes to arguments 205
 floating-point 61
 global 61
 overflow 36
 size 60
 static 61
variant records 170
va_start 138, 205
VAX 60
version 3.0 3
vertical tab character 34, 197
vfprintf 108–112
Video.h 184
void 3, 29
void * 3, 45
void type 45
volatile type qualifier 29, 48–49
vprintf 108–112
vsprintf 108–112

W

%W 162
%w 162
warnings, compile time 41
wchar_t 44, 165
wcstombs 152–153
wctomb 151–152
while statement 51
white space, trailing 116
white-space character 75, 112
windows 13
Windows.h 184
write 131–132
writing Pascal-style functions 56–57
writing to a file 131–132
WR_ONLY 128

X

%X 162

Y

%Y 162
%y 162

Z

%Z 162
Z8000 60

THE APPLE PUBLISHING SYSTEM

This Apple® manual was written, edited, and composed on a desktop publishing system using Apple® Macintosh® computers and Microsoft® Word software. Proof and final pages were created on the Apple LaserWriter® IINTX printer. POSTSCRIPT®, the LaserWriter® page-description language was developed by Adobe Systems Incorporated. The illustrations were created using Adobe Illustrator and some were output to a Linotronic 300.

The illustration on the cover was generated using Adobe Illustrator 88 on a Macintosh® II computer. Some of the images were scanned using an Apple® Scanner and then manipulated in ImageStudio. Initial proofing was done using a QMS color printer. Color separations were done using Adobe separator and output to a Linotronic 300 at standard resolution.

Text type is Apple's corporate font, a condensed version of Garamond. Bullets are ITC Zapf Dingbats®. Some elements, such as programs listings, are set in Apple Courier, a fixed-width font.

Macintosh® Programmer's Workshop C

Version 3.2

This package contains

1	Manual	*Macintosh Programmer's Workshop C*
1	Set of release notes	*MPW® 3.2 C Release Notes*
1	Disk	*MPW C*, Version 3.2
1	1-inch binder	
9	Tab dividers	

If you have any questions, please call

1-800-282-2732 (U.S.)
1-408-562-3910 (International)
1-800-637-0029 (Canada)

M0022LL/D

4/24/91